В серии: Библиотека ALT Linux

ALT Linux изнутри

Дмитрий Аленичев
Александр Боковой
Антон Бояршинов
Вадим Виниченко
Александр Колотов
Георгий Курячий
Дмитрий Левин
Кирилл Маслинский
Максим Отставнов
Алексей Смирнов
Мэтт Уэлш (Matt Welsh)
и другие

ДМК ПРЕСС

Москва, 2006

УДК 004.2
ББК 32.973.26-018.2
А45

А45 ALT Linux снаружи. ALT Linux изнутри:
В серии: «Библиотека ALT Linux». — М.: ALT Linux; Издательский дом ДМК-пресс, 2006. — 416 с.; ил.

ISBN 5-9706-0029-6

Пингвины не вьют гнёзд, а яйца высиживают самцы.

Из чего состоит дистрибутив Linux? Как он работает и почему? Что умеет, и как этими умениями управлять? Кто они такие, эти таинственные «линуксоиды»? пингвины? Если писать программы, то про что их писать? И вообще — что на самом деле такое: «Linux», «Дистрибутив», «Сообщество», «Свободные программы» наконец? Обо всём этом и о некотором другом и написано во второй половине нашей книги.

Раз уж приходится управлять таким сложным созданием, как пингвин, и тем более — таким сложным инструментом, как компьютер, без знаний не обойтись. Если вы не считаете поведение компьютера «предопределением природы», и хотите что-то изменить к лучшему, непременно стоит познакомиться с тем, что именно вы хотите изменить. По крайней мере, мы — разработчики дистрибутивов ALT Linux и участники Linux-сообщества — для себя решили начинать именно со знаний.

В комплект входит дистрибутив ALT Linux Compact 3.0.

Главный редактор: Д. А. Мовчан `dm@dmk-press.ru`

Редактор серии: К. А. Маслинский `kirill@altlinux.ru`

Корректор: М. Л. Романова

Оглавление

К читателю

О чём эта книга

Эта часть книги — о том, чего можно добиться от Linux, в каких случаях он удобен и эффективен, и почему он может быть так привлекателен.

У нас получилось три главных части ответа на эти вопросы: Linux как операционная система, как сообщество и как инструментарий. Мы не пытались исчерпывающим образом изложить все эти три темы, для этого нужна книга совсем другого объёма и основательности. При отборе материала у нас была другая, более важная цель: написать об *интересных* свойствах Linux и их применении на практике.

Пожалуй, изюминка этой половины книги — последний раздел «Своими руками», где приводится несколько жизненных примеров решения довольно сложных задач простыми и стандартными инструментами Linux: автоматизация обработки изображений, создание мультфильмов, обустройство домашней почтовой системы, организация компьютерного класса.

А неизбежную неполноту изложения мы старались где возможно компенсировать ссылками на более подробную информацию. И это, наверное, четвёртая часть ответа на вопрос «А чем же всё-таки хорош этот Linux?» — ничто в нём не скрыто от любознательного, и ответ на любой вопрос обязательно обнаружится, пусть и не под обложкой этой книги.

Как читать эту книгу

В любом порядке, можно даже начать с самого начала и читать подряд до самого конца. Не с меньшим успехом можно читать, начиная с любого раздела, название которого показалось интересным. Что-

бы сделать более удобным чтение в разбивку в конце каждого раздела приведён небольшой список ссылок на другие части книги, в которых найдутся объяснения неизвестных терминов, более подробное или более практичное изложение того же предмета, примеры и т. п. Часть из этих ссылок — на другую половину книги, «ALT Linux снаружи».

А интереснее всего будет, если под рукой окажется Linux, для экспериментов и выяснения подробностей (заметьте, что к этой книге прилагается диск с дистрибутивом ALT Linux 3.0 Compact). В тексте будут периодически встречаться фрагменты, набранные вот таким шрифтом: `command --option /home/user/filename` — это команды, их параметры, имена файлов и других объектов системы, в общем, всё то, что можно ввести с клавиатуры и прочитать с экрана монитора. Не стесняйтесь экспериментировать!

См. также | Обращение к читателю с обратной стороны книги «ALT Linux снаружи» . . . . . [снаружи, стр. 7]

Часть I

Основы ОС Linux

Глава 1

Интерпретатор командной строки (shell)

Георгий Курячий, Кирилл Маслинский

Проницательный читатель, несомненно, заметит, что как только речь заходит об устройстве Linux и более или менее профессиональной работе с этой ОС, в примерах немедленно возникает и начинает доминировать командная строка. Из чего несложно сделать вывод, что это главный (и стандартный) интерфейс управления системой в Linux. Тот же проницательный читатель наверняка задастся вопросом — а кто же выполняет команды, введённые в командной строке? Ответ «система» окажется неправильным: в Linux нет отдельного объекта под именем «система». Система — она на то и система, чтобы состоять из многочисленных компонентов, взаимодействующих друг с другом.

Правильный ответ, как водится, оказывается более сложным. Операционная система нужна в частности для того, чтобы программы можно было писать, не думая о подробностях устройства компьютера и его деталей, начиная от процессора и жёсткого диска (скажем, на уровне «открыть файл», а не последовательности команд перемещения головки жёсткого диска). Операционная система управляет оборудованием сама, а программам предоставляет «язык» довольно высокого уровня абстракции, покрывающий все их необходимости, т. н. **API**[1]. Но для команд пользователя такой язык не годится, поскольку он всё равно слишком низкоуровневый (для решения даже самой простой за-

[1] В Linux основу API составляют **системные вызовы** и стандартные **библиотечные функции**.

дачи пользователя необходимо выполнить несколько таких операций), да и воспользоваться им можно, только написав программу (чаще всего — на языке Си). Возникает необходимость выдумать для пользователя другой — более высокоуровневый и более удобный — язык управления системой. Все команды, которые можно ввести в командной строке, сформулированы именно на этом языке.

Из чего проницательному читателю несложно заключить, что *обрабатывать* эти команды, переводя их на язык операционной системы, должна тоже какая-нибудь специальная программа, и именно с ней ведёт диалог пользователь, работая с командной строкой. Так оно и есть: программа эта называется **интерпретатор командной строки** или **командная оболочка** («shell»). «Оболочкой» она названа как раз потому, что всё управление системой идёт как бы «изнутри» неё: пользователь общается с нею на удобном ему языке (с помощью текстовой командной строки), а она общается с другими частями системы на удобном *им* языке (вызывая запрограммированные функции).

Конечно, командных интерпретаторов в Linux несколько. Самый простой из них, появившийся в ранних версиях UNIX, назывался `sh`, или «Bourne Shell» — по имени автора, Стивена Борна (Stephen Bourne). Со временем его — везде, где только можно — заменили на более мощный, `bash`, «Bourne Again Shell»[1]. bash превосходит `sh` во всём, особенно в возможностях *редактирования* командной строки. Помимо `sh` и `bash` в системе может быть установлен «The Z Shell», `zsh`, *самый* мощный на сегодняшний день командный интерпретатор (шутка ли, 22 тысячи строк документации), или `tcsh`, обновлённая и тоже очень мощная версия старой оболочки «C Shell», синтаксис команд которой похож на язык программирования Си.

Синтаксис командной строки

Итак, что же представляет собой этот более удобный для пользователя язык? Больше всего общение на этом языке напоминает письменный диалог с системой — поочерёдный обмен текстами. Высказывание пользователя на этом языке — это команда, каждая команда — это отдельная строка. Пока не нажат *enter*, строку можно редактировать, затем она передаётся оболочке. Оболочка *разбирает* полученную команду — переводит её на язык системных объектов и функций, после чего отправляет системе на выполнение.

[1] Игра слов: «Bourne Again» вслух читается как «born again», т. е. «возрождённый».

Результат выполнения очень многих команд также представляет собой текст, выдаваемый в качестве «ответа» пользователю. Хотя это и не обязательно — команда может выполнять свою работу совершенно молчаливо. Кроме того, если в процессе выполнения команды возникли какие-то особые обстоятельства (например, ошибка), оболочка включит в ответ пользователю **диагностические сообщения**.

Команда и параметры

Простейшая команда состоит из одного «слова», например, команда `cal`, которая выводит календарь на текущий месяц.

```
[methody@localhost methody]$ cal
    Декабря 2005
Вс Пн Вт Ср Чт Пт Сб
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24

[methody@localhost methody]$
```

А если нужно посмотреть календарь на будущий месяц? Верно, не следует для этого изобретать отдельную команду[1], `cal` вполне справится с этой задачей, только её поведение нужно немного модифицировать:

```
[methody@localhost methody]$ cal 1 2006
    Января 2006
Вс Пн Вт Ср Чт Пт Сб
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
```

[1] Представьте себе язык, в котором для выражения *любой* мысли существует отдельное слово — он был бы невероятно неэффективным, и обязательно нашлась бы мысль, на этом языке невыразимая. В естественных языках для выражения мысли используются мощные средства комбинации и модификации слов, и, соответственно, их значений — грамматика языка. Аналогичный принцип действует и в языке командной оболочки, только здесь «грамматику» принято называть синтаксисом.

Выходит, команда `cal 1 2006` состоит как минимум из двух частей — собственно команды `cal` и «всего остального». Это остальное, что следует за командой, называют **параметрами** (или *аргументами*), причём они вводятся для того, чтобы изменить поведение команды. В большинстве случаев при разборе командной строки *первое* слово считается именем команды, а остальные — её параметрами.

Слова

При разборе командной строки shell использует понятие **разделитель** (delimiter). Разделитель — это символ, разделяющий слова; таким образом командная строка — это последовательность *слов* (которые имеют значение) и *разделителей* (которые значения не имеют). Для shell разделителями являются символ пробела, символ табуляции и символ перевода строки. Количество разделителей между двумя соседними словами значения не имеет.

Для того, чтобы разделитель попал *внутрь* слова (и получившаяся строка с разделителем передалась как *один* параметр), всю нужную подстроку надо окружить одинарными или двойными кавычками:

```
[methody@localhost methody]$ echo One      Two      Three
 One Two Three
[methody@localhost methody]$ echo One      "Two      Three"
 One Two      Three
[methody@localhost methody]$ echo 'One
>
> Ой. И что дальше?
> А, кавычки забыл!'

 One

 Ой. И что дальше?
 А, кавычки забыл!
[methody@localhost methody]$
```

Всё сказанное означает, что у команды столько параметров, сколько *слов* с точки зрения shell следует за ней в командной строке. Первое слово в тройке передаётся команде как первый параметр, второе — как второй и т. д. В первом случае команде `echo` было передано *три* параметра — «`One`», «`Two`» и «`Three`». Она их и вывела, разделяя пробелом. Во втором случае параметров было *два*: «`One`» и «`Two Three`». В результате эти *два* параметра были также выведены через пробел.

В третьем случае параметр был всего *один* — от открывающего апострофа «`'One`» до закрывающего «`...забыл!'`». Всё время ввода `bash` услужливо выдавал подсказку «`> `» — в знак того, что набор командной строки продолжается, но в режиме ввода содержимого кавычек.

Ключи

Для решения разных задач одни и те же действия необходимо выполнять слегка по-разному. Например, для синхронизации работ в разных точках земного шара лучше использовать единое для всех время (по Гринвичу), а для организации собственного рабочего дня — местное время (с учётом сдвига по часовому поясу и разницы зимнего и летнего времени). И то, и другое время показывает команда `date`, только для работы по Гринвичу ей нужен дополнительный параметр «`-u`» (он же «`--universal`»).

```
[methody@localhost methody]$ date
Вск Сен 19 23:01:17 MSD 2004
[methody@localhost methody]$ date -u
Вск Сен 19 19:01:19 UTC 2004
```

Такого рода параметры называются *модификаторами выполнения* или **ключами** (options)[1]. Ключ принадлежит данной конкретной команде и сам по себе смысла не имеет, чем отличается от прочих параметров (например, имён файлов, чисел), которые имеют *собственный* смысл, не зависящий ни от какой команды. Каждая команда может распознавать некоторый набор ключей и соответственно изменить своё поведение. В результате «один и тот же» ключ, например, «`-s`» может значить для разных команд совершенно разные вещи.

Для формата ключей нет жёсткого стандарта, однако существуют договорённости, нарушать которые в наше время уже неприлично. Во-первых, если параметр начинается на «`-`», это — **однобуквенный ключ**. За «`-`», как правило, следует один символ, чаще всего — буква, обозначающая действие или свойство, которое этот ключ придаёт команде. Так проще отличать ключи от других параметров — и пользователю при наборе командной строки, и программисту, автору команды.

Во-вторых, желательно, чтобы имя ключа было *значащим* — как правило, это первая буква названия действия или свойства, обозначаемого ключом. Например, ключ «`-a`» в `man` и `who` происходит от слова

[1] Многие склонны вместо слова «ключ» употреблять слово «опция» как аналог английского option, однако это не признак хорошего стиля.

«**A**ll» (всё), и изменяет работу этих команд так, что они начинают показывать информацию, о которой обычно умалчивают. А в командах `cal` и `who` смысл ключа «`-m`» — разный:

```
[methody@localhost methody]$ who -m
 methody  tty1         Sep 20 13:56 (localhost)
[methody@localhost methody]$ cal -m
     Сентября 2004
 Пн Вт Ср Чт Пт Сб Вс
        1  2  3  4  5
  6  7  8  9 10 11 12
 13 14 15 16 17 18 19
 20 21 22 23 24 25 26
 27 28 29 30
```

Для `who` ключ «`-m`» означает «**M**e», то есть «Я», и в результате `who` работает похоже на `whoami`[1]. А для `cal` ключ «`-m`» — это команда выдать календарь, считая первым днём *понедельник* («**M**onday»), как это принято в России.

Свойство ключа быть, с одной стороны, предельно коротким, а с другой стороны — информативным, называется **аббревиативностью**. Не только ключи, но и имена наиболее распространённых команд Linux обладают этим свойством.

В-третьих, иногда ключ изменяет поведение команды таким образом, что меняется и толкование параметра, следующего в командной строке за этим ключом. Выглядит это так, будто ключ *сам* получает параметр, поэтому ключи такого вида называются **параметрическими**. Как правило, их параметры — имена файлов различного применения, числовые характеристики и прочие *значения*, которые нужно передать команде.

```
[methody@localhost methody]$ date -s 00:00
date: невозможно установить дату: Operation not permitted
Чтв Дек 29 00:00:00 MSK 2005
[methody@localhost methody]$ date
Чтв Дек 29 14:17:38 MSK 2005
```

Ключ «`-s`» команды `date` позволяет установить системное время в то значение, которое указывается в качестве параметра этого ключа. Однако в данном примере эта операция не удалась, о чём свидетельствует выведенное **сообщение об ошибке**. Для изменения системных

[1] Кстати, с незапамятных времён `who` поддерживает один *нестандартный* набор параметров: `who am i` делает то же, что и `who -m`.

часов требуются привилегии системного администратора, а в нашем примере эта команда выполнялась от имени обычного пользователя. Тем не менее, `date` всё равно отобразил время, которое нужно было установить, хотя системные часы и остались не изменёнными.

Аббревиативность ключей трудно соблюсти, когда их у команды *слишком* много. Некоторые буквы латинского алфавита (например, «`s`» или «`o`») используются очень часто, и могли бы служить сокращением сразу нескольких команд, а некоторые (например, «`z`») — редко, под них и название-то осмысленное трудно придумать. На такой случай существует другой, **полнословный** формат: ключ начинается на *два* знака «`-`», за которыми следует *полное* имя обозначаемой им сущности. Таков, например, ключ «`--help`» (аналог «`-h`»):

```
[methody@localhost methody]$ head --help
Использование: head [КЛЮЧ]... [ФАЙЛ]...
Печатает первые 10 строк каждого ФАЙЛА на стандартный вывод.
Если задано несколько ФАЙЛОВ, сначала печатает заголовок с именем
файла.
Если ФАЙЛ не задан или задан как -, читает стандартный ввод.

Аргументы, обязательные для длинных ключей, обязательны и для
коротких.
  -c, --bytes=[-]N         напечатать первые N байт каждого файла;
                             если перед N стоит ‘-’, напечатать
все, кроме N
                             последних байт каждого файла
  -n, --lines=[-]N         напечатать первые N строк каждого
файла, а не 10;
                             если перед N стоит ‘-’, напечатать
все, кроме N
                             последних строк каждого файла
  -q, --quiet, --silent    не печатать заголовки с именами файлов
  -v, --verbose            всегда печатать заголовки с именами
файлов
      --help     показать эту справку и выйти
      --version  показать информацию о версии и выйти


N может иметь суффикс-множитель: b 512, k 1024, m 1024*1024.

Об ошибках сообщайте по адресу <bug-coreutils@gnu.org>.
```

Видно, что некоторые ключи `head` имеют и однобуквенный, и полнословный формат, а некоторые — только полнословный. Так обычно и бывает: часто используемые ключи имеют аббревиатуру, а редкие — нет. *Значения* параметрических *полнословных* ключей принято передавать не следующим параметром командной строки, а с помощью конструкции «`=значение`» непосредственно после ключа.

В-четвёртых, есть некоторые менее жёсткие, но популярные договорённости о *значении* ключей. Ключ «`-h`» («**H**elp») обычно (но, увы, не всегда) заставляет команды выдать *краткую справку*. Наконец, бывает необходимо передать команде *параметр, а не ключ*, начинающийся с «-». Для этого нужно использовать ключ «`--`»:

```
[methody@localhost methody]$ head -1 -filename-with-
head: invalid option -- f
Попробуйте ‘head --help’ для получения более подробного описания.
[methody@localhost methody]$ head -1 -- -filename-with-
Первая строка файла -filename-with-
```

Ключ «`--`» (первый «-» — признак ключа, второй — сам ключ) обычно запрещает команде интерпретировать все последующие параметры командной строки как ключи, независимо от того, начинаются ли они на «-» или нет. Только после «`--`» `head` согласилась с тем, что `-filename -with-` — это имя файла.

Синопсис

Из всего вышесказанного ясно, что для каждой команды существует свой собственный небольшой язык — его составляют те ключи и обязательные и необязательные параметры, которые принимает и интерпретирует команда. Чтобы окинуть возможности команды одним взглядом, в различной документации по Linux приводится **синопсис** — сжатое перечисление всех возможных параметров команды. Выглядит это примерно так:

```
cal [-smjy13] [[month] year]
```

В синопсисе даётся *формализованное* описание способов использования объекта (в данном случае — того, как и с какими параметрами запускать команду `cal`). Параметры перечисляются в том же порядке, в котором их нужно вводить в командной строке, необязательные параметры, как правило, даются в квадратных скобках, обязательные — вообще без скобок, а ключи для компактности собираются в один параметр, в котором каждая буква — это отдельный ключ. Приведённый

пример читается так: у команды `cal` нет обязательных параметров, есть (необязательные) ключи (`-s`, `-m` и т. д.), и необязательный параметр `year`, перед которым может присутствовать необязательный же `month` (но не может быть указан `month` без `year`).

Откуда берутся команды

Дочитав предыдущий раздел, проницательный читатель должен был подумать примерно так: ага, ну с командами и параметрами (т. е. с грамматикой командной строки) мы немного разобрались, вооружите же нас теперь списком всех команд Linux (иначе говоря, словарём), и мы примемся за работу. Почему же нигде не напечатан такой список? Точнее, списков команд много разных и все они очевидно неполные и не во всём сходятся. Ответ на этот вопрос состоит из двух частей.

Часть 1: команды и утилиты

Shell, командный интерпретатор, является «оболочкой» не только для пользователя, но и для команд: сам он почти никакие команды не исполняет, передаёт системе. Его задача сводится к тому, чтобы разобрать командную строку, выделить из неё команду и параметры, а затем запустить **утилиту**[1] — программу, имя которой совпадает с именем команды.

Если смотреть «изнутри» командного интерпретатора, то работа с командной строкой происходит примерно так: пользователь вводит строку (команду), shell считывает её, иногда — преобразует по определённым правилам, получившуюся строку разбивает на команду и параметры, а затем запускает утилиту, передавая ей эти параметры. Утилита, в свою очередь, анализирует параметры, выделяет среди них ключи и делает, что попросили, попутно выводя данные для пользователя, после чего завершается. По завершении утилиты возобновляется работа «отступившего на задний план» командного интерпретатора, он снова считывает командную строку, разбирает её, вызывает команду... Так продолжается до тех пор, пока пользователь не скомандует оболочке *завершиться* самой (с помощью команды `logout` или управляющего символа *Ctrl+D*).

[1] Все программы, которые здесь обсуждаются и будут обсуждаться, принято называть утилитами, то есть «полезными программами».

Однако часть команд (меньшую) оболочка всё же выполняет самостоятельно, не вызывая никаких утилит. Некоторые — самые нужные — команды встроены в `bash`, даже несмотря на то, что они имеются в виде утилит (например, `echo`). Работает встроенная команда так же, но так как времени на её выполнение уходит существенно меньше, командный интерпретатор выберет именно её, если будет такая возможность. В `bash` тип команды можно определить с помощью команды `type`. Собственные команды `bash` называются **builtin** (встроенная команда), а для утилит выводится **путь**, содержащий название каталога, в котором лежит файл с соответствующей программой, и имя этой программы. Ключ «`-a`» («**a**ll», конечно), заставляет `type` вывести *все* возможные варианты интерпретации команды, а ключ «`-t`» — вывести тип команды вместо пути.

```
[methody@localhost methody]$ type date
 info is /bin/date
[methody@localhost methody]$ type echo
 echo is a shell builtin
[methody@localhost methody]$ type -a echo
 echo is a shell builtin
 echo is /bin/echo
[methody@localhost methody]$ type -a -t echo
 builtin
 file
```

Собственных команд в командном интерпретаторе немного. В основном это — операторы языка программирования и прочие средства управления самим интерпретатором. Все команды, выполняющие содержательную работу для пользователя, представлены в Linux в виде отдельных утилит. Вот и первая часть ответа на вопрос обо всех командах Linux: их столько же, сколько есть программ (утилит), написанных для Linux. Их список — это список установленных в системе утилит, и в разных системах он будет различным.

Часть 2: всему своё руководство

Каждый объект системы: все **утилиты**, все **демоны** Linux, все функции **ядра** и **библиотек**, структура большинства **конфигурационных файлов**, наконец, многие умозрительные, но важные понятия — должны обязательно сопровождаться документацией, описывающей их назначение и способы использования. Поэтому от пользователя системы не требуется *заучивать* все возможные варианты

взаимодействия с ней. Достаточно *понимать* основные принципы её устройства и уметь находить справочную информацию. Эйнштейн говорил на этот счёт так: «Зачем запоминать то, что всегда можно посмотреть в справочнике?».

Больше всего *различной* полезной информации содержится в **страницах руководства** (**manpages**). Каждая страница руководства (для краткости — просто «руководство») посвящена какому-нибудь одному объекту системы. Для того, чтобы посмотреть страницу руководства, нужно дать команду `man объект`:

```
[methody@localhost methody]$ man cal
 CAL(1)               BSD General Commands Manual
CAL(1)

 NAME
     cal - displays a calendar

 SYNOPSIS
     cal [-smjy13] [[month] year]

 DESCRIPTION
     Cal displays a simple calendar.  If arguments are not
specified,
         the current month is displayed.  The options are as
follows:
 . . .
```

«Страница руководства» занимает, как правило, больше одной страницы *экрана*. Для того, чтобы читать было удобнее, `man` запускает программу постраничного просмотра текстов — `less`. Управлять программой `less` просто: страницы перелистываются пробелом, а когда читать надоест, надо нажать «`q`» (Quit). Перелистывать страницы можно и клавишами *Page Up*/*Page Down*, для сдвига на *одну* строку вперёд можно применять *enter* или стрелку вниз, а на одну строку назад — стрелку вверх. Переход на начало и конец текста выполняется по командам «`g`» и «`G`» соответственно (Go). Полный список того, что можно делать с текстом в `less`, выводится по команде «`H`» (Help).

Страница руководства состоит из **полей** — стандартных разделов, с разных сторон описывающих объект. При первом изучении руководства стоит начать с полей `NAME` (краткое описание объекта) и `DESCRIPTION` (развёрнутое описание объекта, достаточное для того, чтобы им воспользоваться). Одно из самых важных полей руковод-

ства находится в конце текста. Если в процессе чтения NAME или DESCRIPTION пользователь понимает, что не нашёл в руководстве того, что искал, он может захотеть посмотреть, а есть ли *другие* руководства или иные источники информации *по той же теме*. Список таких источников содержится в поле SEE ALSO:

```
[methody@localhost methody]$ man man
 . . .
 SEE ALSO
        apropos(1), whatis(1), less(1), groff(1), man.conf(5).
 . . .
```

В поле SEE ALSO обнаружились ссылки на руководства по less, groff (программе форматирования страницы руководства), структуре **конфигурационного файла** для man, а также по двум сопутствующим командам whatis и apropos, которые помогают отыскать нужное руководство. Обе они работают с базой данных, состоящей из полей NAME всех страниц руководства в системе. Различие между ними — в том, что whatis ищет только среди имён объектов (в *левых* частях полей NAME), а apropos — по всей базе. В результате у whatis получается список кратких описаний объектов с *именами*, включающими в себя искомое слово, а у apropos — список, в котором это слово *упоминается*.

В системе может встретиться несколько объектов *разного* типа, но с одинаковым названием. Часто совпадают, например, имена **системных вызовов** (функций **ядра**) и утилит, которые позволяют пользоваться этими функциями из командной строки. При ссылке на руководство по объекту системы принято непосредственно после имени объекта ставить в круглых скобках номер раздела, в котором содержится руководство по этому объекту: man(1), less(1), passwd(5). По такому формату легко опознать, что имеется в виду руководство.

В системе руководств Linux девять разделов, каждый из которых содержит страницы руководства к объектам определённого типа. Все разделы содержат по одному руководству с именем «intro», в котором в общем виде и на примерах рассказано, что за объекты имеют отношение к данному разделу. Список разделов с названиями можно получить командой whatis intro.

По умолчанию man просматривает все разделы и показывает *первое найденное* руководство с заданным именем. Чтобы посмотреть руководство по объекту из определённого раздела, необходимо в качестве первого параметра команды man указать номер раздела, например, man 8 passwd.

Другой источник информации о Linux и составляющих его программах — справочная подсистема info. Страница руководства, несмотря на обилие ссылок различного типа, остаётся «линейным» текстом, структурированным только логически. Документ info — это настоящий гипертекст, в котором множество небольших страниц объединены в дерево. В каждом разделе документа info всегда есть оглавление, из которого можно перейти сразу к нужному подразделу, откуда всегда можно вернуться обратно. Кроме того, info-документ можно читать и как *непрерывный* текст, поэтому в каждом подразделе есть ссылки на предыдущий и последующий подразделы. Можно догадаться, что подробное руководство по тому, как перемещаться между страницами в info можно получить по команде info info. Команда info, введённая без параметров, предлагает пользователю список всех документов info, установленных в системе.

Если некоторый объект системы не имеет документации ни в формате man, ни в формате info, это нехорошо. В этом случае можно надеяться, что при нём есть *сопроводительная документация*, не имеющая, увы, ни стандартного формата, ни тем более — ссылок на руководства по другим объектам системы. Такая документация (равно как и примеры использования объекта), обычно помещается в каталог /usr/share/doc/имя_объекта.

Документация в подавляющем большинстве случаев пишется на простом английском языке. Если английский — не родной язык для автора документации, она будет только проще. Традиция писать по-английски идёт от немалого вклада США в развитие компьютерной науки вообще и Linux в частности. Кроме того, английский становится языком международного общения во всех областях, не только в компьютерной. Необходимость писать на языке, который будет более или менее понятен большинству пользователей, объясняется постоянным развитием Linux. Дело не в том, что страницу руководства нельзя перевести, а в том, что её придётся переводить *всякий раз*, когда изменится описываемый ею объект! Например, выход новой версии программного продукта сопровождается изменением его возможностей и особенностей работы, а следовательно, и новой версией документации. Тогда *перевод* этой документации превращается в «moving target», сизифов труд.

Тем не менее, некоторые наиболее актуальные руководства всё-таки существуют в переводе на русский язык. Наиболее свежие версии таких переводов на русский собраны в пакете man-pages-ru —

достаточно установить этот пакет, и те руководства, для которых есть перевод, `man` будет по умолчанию отображать на русском языке.

Переменные окружения

Помимо параметров, передаваемых в командной строке, в Linux есть ещё один способ модифицировать поведение программы — для этого используются **переменные окружения**. Чтобы объяснить принцип работы переменных окружения, потребуется небольшой экскурс в механизм взаимодействия процессов в Linux.

Выполняющаяся программа называется в Linux **процессом**. Каждый запускаемый процесс система снабжает неким информационным пространством, которое этот процесс вправе изменять как ему заблагорассудится — это и есть **окружение** (по-английски environment). Правила пользования этим пространством просты: в нём можно задавать именованные хранилища данных (**переменные окружения**), в которые записывать какую угодно информацию (присваивать значение переменной окружения), а впоследствии эту информацию считывать (подставлять значение переменной).

Процессы — это основные действующие лица в системе. Когда пользователь отдаёт команды в командной строке, то новые процессы для выполнения этих команд (внешние утилиты и т. п.) запускает другой процесс — тот самый командный интерпретатор, который общается с пользователем и принимает от него команды.

Создание одного процесса другим называется *порождением процесса* и происходит в два этапа: сначала создаётся точная копия исходного, *родительского* процесса (системный вызов `fork()`), а затем копия процесса подменяется новым, *дочерним* (системный вызов `exec()`). Для нас сейчас важно, что при этой подмене сохраняются все свойства исходного процесса, и, в частности, окружение.

Вернёмся к работе командного интерпретатора: выполняя команду, он запускает нужную утилиту в качестве дочернего процесса, дожидается окончания её работы (при помощи ещё одного системного вызова, `wait()`), анализирует результат и продолжает работу. Запущенная утилита получает от родительского процесса (командного интерпретатора) информацию двух типов: *параметры командной строки* (не в том виде, в котором их ввёл пользователь, а после обработки по правилам командного интерпретатора, в виде последовательного списка)

и *окружение*, то есть все переменные и их значения, которые были определены в окружении родительского процесса.

Одна и та же утилита может быть использована *одним и тем же* способом, но в изменённом окружении — и выдавать различные результаты. Пользователь может явно изменить окружение для запускаемого процесса, присвоив некоторое значение переменной окружения в командной строке *перед* именем команды. Командный интерпретатор, увидев «=» внутри первого слова командной строки, приходит к выводу, что это — операция присваивания, а не имя команды, и запоминает, как надо изменить окружение команды, которая последует после.

```
[methody@localhost methody]$ date
 Птн Ноя  5 16:20:16 MSK 2004
[methody@localhost methody]$ LC_TIME=C date
 Fri Nov  5 16:20:23 MSK 2004
```

Переменная окружения `LC_TIME` предписывает использовать определённый язык при выводе даты и времени, а значение «`C`» соответствует «стандартному системному» языку (чаще всего — английскому).

Переменные, которые командный интерпретатор `bash` определяет *после* запуска, не принадлежат окружению, и, стало быть, не наследуются дочерними процессами. Чтобы переменная `bash` попала в окружение, её надо *проэкспортировать* командой `export`:

```
[methody@localhost methody]$ LC_TIME=C
[methody@localhost methody]$ date
 Птн Ноя  5 16:20:16 MSK 2004
[methody@localhost methody]$ export LC_TIME=C
[methody@localhost methody]$ date
 Fri Nov  5 16:20:23 MSK 2004
```

Во время сеанса работы пользователя командный интерпретатор получает довольно богатое окружение, к которому добавляет и собственные настройки. Большинство заранее определённых переменных используются либо самой командной оболочкой, либо утилитами системы, поэтому их изменение приводит к тому, что оболочка или утилиты начинают работать слегка иначе. Просмотреть окружение в `bash` можно с помощью команды `set`.

К значению любой переменной в `bash` можно обратиться по имени: вместо конструкции `$имя_переменной` оболочка подставит значение этой переменной. Например, для того, чтобы просмотреть значение

некоторой переменной, пользователь может ввести команду `echo $имя_переменной`.

```
[methody@localhost methody] echo $PATH
/home/methody/bin:/bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin:
/usr/games
```

Весьма примечательна переменная окружения `PATH`. В ней содержится список каталогов, элементы которого разделяются двоеточиями. Если команда в командной строке — не собственная команда shell (вроде `cd`) и не представлена в виде *пути* к запускаемому файлу (как `/bin/ls` или `./script`), то shell будет искать эту команду среди имён запускаемых файлов во всех каталогах `PATH`, и только в них. По этой причине исполняемые файлы невозможно запускать просто по имени, если они лежат в текущем каталоге, и текущий каталог не входит в `PATH`. В таких случаях можно воспользоваться кратчайшим из возможных путей, «`./`» (например, вызывая сценарий `./script`).

Переменных окружения, влияющих на работу *разных* утилит, довольно много. Например, переменные семейства `LC_` (полный их список выдаётся командой `locale`), определяющие язык, на котором выводятся диагностические сообщения, стандарты на формат даты, денежных единиц, чисел, способы преобразования строк и т. п. Если какая-то утилита требует редактирования файла, этот файл передаётся программе, путь к которой хранится в переменной `EDITOR` (обычно это `/usr/bin/vi`), а если потребуется открыть html-файл, то многие утилиты вызовут для этого программу, указанную в переменной `BROWSER`. Наконец, некоторые переменные вроде `UID`, `USER` или `PWD` просто содержат полезную информацию, которую можно было бы добыть и другими способами.

См. также

Перенаправление ввода и вывода . . . . . [стр. 112]

Различные удобства при работе с командной строкой [стр. 118]

Эффективность командной оболочки: скрипты [стр. 156]

Возможности командной строки: создание мультфильмов . . . . . [стр. 163]

Глава 2

Файлы

Файловая система Linux

Кирилл Маслинский

Операционные системы хранят данные на диске при помощи **файловых систем**. Классическая файловая система представляет данные в виде вложенных друг в друга **каталогов** (их ещё называют папками), в которых содержатся **файлы**[1]. Один из каталогов является «вершиной» файловой системы (а выражаясь технически — «корнем»[2]), в нём содержатся (или, если угодно, из него растут) все остальные каталоги и файлы.

Если жёсткий диск разбит на разделы, то на *каждом* разделе организуется отдельная файловая система с собственным корнем и структурой каталогов (ведь разделы полностью изолированы друг от друга).

В Linux корневой каталог называется весьма лаконично — «`/`». Полные имена (пути) всех остальных каталогов получаются из «`/`», к которому дописываются справа имена последовательно вложенных друг в друга каталогов. Имена каталогов в пути также разделяются символом «`/`» («слэш»). Например, запись `/home` обозначает каталог «`home`» в корневом каталоге («`/`»), а `/home/user` — каталог «`user`» в каталоге «`home`» (который, в свою очередь, в корневом каталоге)[3]. Перечисленные таким образом каталоги, завершающиеся именем файла составляют **полный путь** к файлу.

[1] Файл — область данных, имеющая собственное имя.

[2] Такой каталог называют **корневым каталогом**, поскольку он служит корнем дерева файловой системе (в математическом смысле слов «дерево» и «корень»).

[3] Весьма похожий способ записи полного пути используется в системах DOS и Windows, с той разницей, что корневой каталог обозначается литерой устройства с последующим двоеточием, а в качестве разделителя используется символ «\» («обратный слэш»).

Относительный путь строится точно так же, как и полный — перечислением через «/» всех названий каталогов, встретившихся при движении к искомому каталогу или файлу. Между полным путём и относительным есть только одно существенное различие: относительный путь начинается *от текущего каталога*, в то время как полный путь всегда начинается *от корневого каталога*. Относительный путь любого файла или каталога в файловой системе может иметь любую конфигурацию: чтобы добраться до искомого файла можно двигаться как по направлению к корневому каталогу, так и от него. Linux различает полный и относительный пути очень просто: если имя объекта *начинается* на «/» — это полный путь, в любом другом случае — относительный.

Монтирование

Корневой каталог в Linux всегда только *один*, а все остальные каталоги в него вложены, т. е. для пользователя файловая система представляет собой единое целое[1]. В действительности, разные части файловой системы могут находиться на совершенно разных устройствах: разных разделах жёсткого диска, на разнообразных съёмных носителях (лазерных дисках, дискетах, флэш-картах), даже на других компьютерах (с доступом через сеть). Для того, чтобы соорудить из этого хозяйства единое дерево с одним корнем, используется процедура **монтирования**.

Монтирование — это подключение в один из каталогов целой файловой системы, которая находится где-то на другом устройстве. Эту операцию можно представить как «прививание» ветки к дереву. Для монтирования необходим пустой каталог — он называется **точкой монтирования**. Точкой монтирования может служить любой каталог, никаких ограничений на этот счёт в Linux нет. При помощи специальной команды (`mount`) мы объявляем, что в данном *каталоге* (пока пустом) нужно отображать файловую систему, доступную на таком-то *устройстве* или же по сети. После этой операции в каталоге (точке монтирования) появятся все те файлы и каталоги, которые находятся на соответствующем устройстве. В результате пользователь может даже и не знать, на каком устройстве какие файлы располагаются.

[1]Это отличается от технологии, применяемой в Windows или Amiga, где для каждого устройства, на котором есть файловая система, используется свой корневой каталог, обозначенный литерой, например «`a`», «`c`», «`d`» и т. д.

Подключённую таким образом («смонтированную») файловую систему можно в любой момент отключить — **размонтировать** (для этого имеется специальная команда `umount`), после чего тот каталог, куда она была смонтирована, снова окажется пустым.

Одно из устройств для Linux является самым важным — это **корневая файловая система** (root filesystem). Именно к ней затем будут подключаться (монтироваться) все остальные файловые системы на других устройствах. Обратите внимание, что корневая файловая система тоже монтируется, но только не к другой файловой системе, а к «самой Linux», причём точкой монтирования служит «/» (корневой каталог). Поэтому при загрузке системы прежде всего монтируется корневая файловая система, а при останове она размонтируется (в последнюю очередь).

Пользователю обычно не требуется выполнять монтирование и размонтирование вручную: при загрузке системы будут смонтированы все устройства, на которых хранятся части файловой системы, а при останове (перед выключением) системы все они будут размонтированы. Файловые системы на съёмных носителях (лазерных дисках, дискетах и пр.) также монтируются и размонтируются автоматически — либо при подключении носителя, либо при обращении к соответствующему каталогу.

Стандартные каталоги

В корневом каталоге Linux-системы обычно находятся только подкаталоги со *стандартными* именами. Более того, не только имена, но и *тип данных*, которые могут попасть в тот или иной каталог, также регламентированы стандартом[1]. Этот стандарт довольно последовательно соблюдается во всех Linux-системах: так, в любой Linux вы всегда найдёте каталоги `/etc`, `/home`, `/usr/bin` и т. п. и сможете довольно точно предсказать, что именно в них находится.

Стандартное размещение файлов позволяет и человеку, и даже программе предсказать, где находится тот или иной компонент си-

[1]Этот стандарт называется **Filesystem Hierarchy Standard** («стандартная структура файловых систем»). Стандарт **FHS** регламентирует не только перечисленные каталоги, но и их подкаталоги, а иногда даже приводит список конкретных файлов, которые должны присутствовать в определённых каталогах. Краткое описание стандартной иерархии каталогов Linux можно получить, отдав команду `man hier`. Полный текст и последнюю редакцию стандарта **FHS** можно найти в пакете `fhs` или прочесть по адресу `http://www.pathname.com/fhs/`.

стемы. Для человека это означает, что он сможет быстро сориентироваться в любой системе Linux (где файловая система организована в соответствии со стандартом) и найти то, что ему нужно. Для программ стандартное расположение файлов — это возможность организации автоматического взаимодействия между разными компонентами системы.

См. также | Разделы жёсткого диска . . . . [снаружи, стр. 185]

Работа с файлами

Георгий Курячий, Кирилл Маслинский

Текущий каталог

Файловая система не только систематизирует данные, но и является основой метафоры «рабочего места» в Linux. Каждая выполняемая программа «работает» в строго определённом каталоге файловой системы. Такой каталог называется **текущим каталогом**, можно представлять, что программа во время работы «находится» именно в этом каталоге, это её «рабочее место». В зависимости от текущего каталога может меняться поведение программы: зачастую программа будет по умолчанию работать с файлами, расположенными именно в текущем каталоге — до них она «дотянется» в первую очередь. Текущий каталог есть у любой программы, в том числе и у командной оболочки (shell) пользователя. Поскольку взаимодействие пользователя с системой обязательно опосредовано командной оболочкой, можно говорить о том, что пользователь «находится» в том каталоге, который в данный момент является *текущим каталогом его командной оболочки*.

Все команды, отдаваемые пользователем при помощи shell, наследуют текущий каталог shell, т. е. «работают» в том же каталоге. По этой причине пользователю важно знать текущий каталог shell. Для этого служит утилита `pwd`:

```
[methody@localhost methody]$ pwd
/home/methody
[methody@localhost methody]$
```

`pwd` (аббревиатура от **p**rint **w**orking **d**irectory) возвращает полный путь текущего каталога командной оболочки, естественно, именно той

командной оболочки, при помощи которой была выполнена команда `pwd`. В данном примере текущим является каталог «`/home/methody`».

Текущий каталог, каков бы ни был полный путь к нему, всегда имеет ещё одно обозначение, «`.`», которое можно использовать, если по каким-то причинам требуется, чтобы даже в *относительном* пути к файлу, находящемуся в *текущем* каталоге, присутствовал элемент «имя каталога». Так, пути «`text`» и «`./text`» тоже приводят к одному и тому же файлу, однако в первом случае в строке пути не содержится ничего, кроме имени файла.

Отделить путь к файлу от его имени можно с помощью команд `dirname` и `basebane` соответственно:

```
[methody@localhost methody]$ basename /home/methody/text
text
[methody@localhost methody]$ basename text
text
[methody@localhost methody]$ dirname /home/methody/text
/home/methody
[methody@localhost methody]$ dirname ./text
.
[methody@localhost methody]$ dirname text
.
```

Заметьте, что для «`text`» и «`./text`» `dirname` выдало одинаковый результат: «`.`», что понятно: как было сказано выше, эти формы пути совершенно эквивалентны, а при *автоматической* обработке результатов `dirname` гораздо лучше получить «`.`», чем *пустую строку*.

Информация о каталоге

В любой момент можно просмотреть содержимое любого каталога при помощи утилиты `ls` (сокращение от англ. «list» — «список»):

```
[methody@localhost methody]$ ls
-filename-with- text
[methody@localhost methody]$
```

Поданная без параметров, команда `ls` выводит список файлов и каталогов, содержащихся в *текущем каталоге*[1].

Утилита `ls` принимает один **параметр**: имя каталога, содержимое которого нужно вывести. Имя может быть задано любым доступным

[1] Вот пример утилиты, которая по умолчанию работает с файлами в текущем каталоге.

способом: в виде **полного** или **относительного пути**. Кроме параметра, утилита `ls` «понимает» множество ключей, которые нужны главным образом для того, чтобы выводить дополнительную информацию о файлах в каталоге или выводить список файлов выборочно. Чтобы узнать обо всех возможностях `ls`, нужно, конечно же, прочесть **руководство** по этой утилите («`man ls`»).

```
[methody@localhost methody]$ ls -F /
bin/   dev/  home/  mnt/   root/  swap/  tmp/  var/
boot/  etc/  lib/   proc/  sbin/  sys/   usr/
[methody@localhost methody]$
```

В примере использован ключ «`-F`», чтобы отличать файлы от каталогов. При наличии этого ключа `ls` в конце имени каждого каталога ставит символ «/», чтобы показать, что в нём может содержаться что-то ещё. В выведенном списке нет ни одного файла — в корневом каталоге содержатся только подкаталоги.

Кроме того, можно получить более подробную информацию о содержимом каталога:

```
[methody@localhost methody]$ ls -aF
-filename-with-  .bash_history  .bashrc  .lpoptions  .rpmmacros
Documents/
./               .bash_logout   .emacs   .mutt/      .xemacs/
text
../              .bash_profile  .i18n    .pinerc     .xsession.d/
tmp/
[methody@localhost methody]$
```

Внезапно обнаружилось, что файлов в домашнем каталоге не два, а гораздо больше. Дело в том, что утилита `ls` по умолчанию не выводит информацию об объектах, чьё имя начинается с «.» — в том числе о «.» и «..». Для того, чтобы посмотреть *полный* список содержимого каталога, и используется ключ «`-a`» (**a**ll)[1]. Как правило, с «.» начинаются имена **конфигурационных файлов** и **конфигурационных каталогов**.

«..» — это ссылка на **родительский каталог**. Родительский каталог — это тот каталог, *в котором* находится данный. Родительским

[1]Такое поведение `ls` напоминает принцип работы файловых менеджеров со скрытыми файлами в системах dos/win. Разница в том, что в dos/win скрытые файлы предусмотрены файловой системой — файл может иметь *атрибут* «скрытый» и при этом называться как угодно. В Linux скрытые файлы — это не свойство файловой системы, а только *соглашение по наименованию* файлов.

каталогом для «`/home/methody`» будет каталог «`/home`»: он получается просто отбрасыванием последнего имени каталога в полном пути. Иначе можно сказать, что родительский каталог — это один шаг по дереву каталогов по направлению к корню. «..» — это сокращённый способ сослаться на родительский каталог: пока текущим каталогом является «`/home/methody`», **относительный путь** «..» (или, что то же самое, «./..») будет эквивалентен «`/home`». С использованием «..» можно строить *сколь угодно* длинные пути, такие как «`../../usr/../var/log/../run/../../home`»[1]. Ссылки на текущий и на родительский каталог обязательно присутствуют в *каждом* каталоге в Linux. Даже если каталог пуст, т. е. не содержит ни одного файла или подкаталога, команда «`ls -a`» выведет список из двух имён: «.» и «..».

Перемещение по дереву каталогов

Пользователь может работать с файлами не только в своём домашнем каталоге, но и в других каталогах. В этом случае будет удобно *сменить* **текущий каталог**, т. е. «переместиться» в другую точку файловой системы. Для смены текущего каталога командной оболочки используется команда `cd` (от англ. «change directory» — «сменить каталог»). Команда `cd` принимает один параметр: имя каталога, в который нужно переместиться — сделать текущим. Как обычно, в качестве имени каталога можно использовать полный или относительный путь.

```
[methody@localhost methody]$ cd /home
[methody@localhost home]$ ls
methody  shogun
[methody@localhost home]$ cd methody
[methody@localhost methody]$
```

Для перемещения в родительский каталог («`/home`») удобно воспользоваться ссылкой «..». Необходимость вернуться в домашний каталог из произвольной точки файловой системы возникает довольно часто, поэтому командная оболочка поддерживает обозначение домашнего каталога при помощи символа «~». Поэтому чтобы перейти в домашний каталог из любого другого, достаточно выполнить команду «`cd ~`». При исполнении команды символ «~» будет заменён командной оболочкой на полный путь к домашнему каталогу пользователя.

[1]Не сразу понятно, что приводит этот путь всё туда же, в «`/home`».

```
[methody@localhost methody]$ cd ..
[methody@localhost home]$ cd ~
[methody@localhost methody]$ cd ~shogun
[methody@localhost shogun]$ cd
[methody@localhost methody]$
```

При помощи символа «~» можно ссылаться и на домашние каталоги других пользователей: «**~имя пользователя**». Команда `cd`, поданная без параметров, эквивалента команде «`cd ~`» и делает текущим каталогом домашний каталог пользователя.

Создание каталогов

Для этого используется утилита `mkdir`. Она используется с одним обязательным параметром: именем создаваемого каталога. По умолчанию каталог будет создан в текущем каталоге.

```
[methody@localhost methody]$ mkdir examples
[methody@localhost methody]$ ls -F
-filename-with-  Documents/  examples/  text  tmp/
[methody@localhost methody]$
```

Копирование и перемещение файлов

Для перемещения файлов и каталогов предназначена утилита `mv` (сокращение от англ. «move» — «перемещать»). У `mv` два обязательных параметра: первый — перемещаемый файл или каталог, второй — файл или каталог назначения. Имена файлов и каталогов могут быть заданы в любом допустимом виде: при помощи полного или относительного пути. Кроме того, `mv` позволяет перемещать не только один файл или каталог, а сразу несколько. За подробностями о допустимых параметрах и ключах следует обратиться к руководству по `mv`.

```
[methody@localhost methody]$ cd examples
[methody@localhost examples]$ mv ../text .
[methody@localhost examples]$ ls
text
[methody@localhost examples]$
```

Перемещение файла внутри одной файловой системы в действительности равнозначно его *переименованию*: данные самого файла при этом остаются на тех же секторах диска, изменяются *каталоги*, в которых произошло перемещение. Перемещение предполагает

удаление ссылки на файл из того каталога, откуда он перемещён, и добавление ссылки на этот самый файл в тот каталог, куда он перемещён. В результате изменяется полное имя файла — **полный путь**, т. е. положение файла в файловой системе.

Иногда требуется создать копию файла: для бóльшей сохранности данных, для того, чтобы создать модифицированную версию файла и т. п. В Linux для этого предназначена утилита `cp` (сокращение от англ. «copy» — «копировать»). Утилита `cp` требует присутствия двух обязательных параметров: первый — копируемый файл или каталог, второй — файл или каталог назначения. Как обычно, в именах файлов и каталогов можно использовать полные и относительные пути. Есть несколько возможностей при комбинации файлов и каталогов в параметрах `cp` — о них можно прочесть в **руководстве**.

```
[methody@localhost examples]$ cp text text.bak
[methody@localhost examples]$ ls
text  text.bak
```

Нужно иметь в виду, что в Linux утилита `cp` нередко настроена таким образом, что при попытке скопировать файл поверх уже существующего не выводится никакого предупреждения. В этом случае файл будет просто перезаписан, а данные, которые содержались в старой версии файла, бесповоротно потеряны. Поэтому при использовании `cp` следует всегда быть внимательным и проверять имена файлов, которые нужно скопировать.

Удаление файлов и каталогов

В Linux для удаления файлов предназначена утилита `rm` (сокращение от англ. «remove» — «удалять»).

```
[methody@localhost methody]$ rm examples/text
[methody@localhost methody]$ ls examples
test.bak
```

Однако удалить командой `rm` каталог не получится:

```
[methody@localhost methody]$ rm examples
rm: невозможно удалить ‘examples’: Is a directory
[methody@localhost methody]$ rmdir examples
rmdir: examples: Directory not empty
[methody@localhost methody]$ rm examples/test.bak
[methody@localhost methody]$ rmdir examples
[methody@localhost methody]$
```

Область подкачки используется в Linux для организации виртуальной памяти: когда программам недостаточно имеющейся в наличии оперативной памяти, часть рабочей информации временно размещается на жёстком диске.

`JFS`

Разработана IBM для файловых серверов с высокой нагрузкой: при разработке особый упор делался на производительность и надёжность, что и было достигнуто. Поддерживает журнализацию.

В Linux поддерживается, кроме собственных, немало форматов файловых систем, используемых другими ОС. Если способ записи на эти файловые системы *известен* и не слишком замысловат, то работает и запись, и чтение, в противном случае — только чтение (чего нередко бывает достаточно). Файловые системы перечисленных ниже типов обычно присутствуют на разделах диска, принадлежащих другим операционным системам.

`FAT12/FAT16/FAT32`

Эти файловые системы используются в MS-DOS и разных версиях Windows, а также на многих съёмных носителях (в частности, на дискетах и USB-flash). Linux поддерживает чтение и запись на эти файловые системы.

`NTFS`

Файловая система NTFS изначально появилась в системах Windows NT, но может использоваться и другими версиями Windows (например, Windows 2000). В Linux NTFS поддерживается только на чтение.

См. также | Планирование жёсткого диска . [снаружи, стр. 189]

Глава 3

Права

Пользователи в Linux

Кирилл Маслинский

Учётные записи

Linux — система многопользовательская, а потому пользователь — ключевое понятие для организации всей системы доступа в Linux. Когда пользователь регистрируется в системе (проходит процедуру авторизации, например, вводя системное имя и пароль), он идентифицируется с **учётной записью**, в которой система хранит информацию о каждом пользователе: его системное имя и некоторые другие сведения, необходимые для работы с ним. Именно с учётными записями, а не с самими пользователями, и работает система. Ниже приведён список этих сведений.

Системное имя (user name)

Это то имя, которое вводит пользователь в ответ на приглашение `login:`. Оно может содержать только латинские буквы и знак «_». Это имя используется также в качестве имени учётной записи.

Идентификатор пользователя (UID)

Linux связывает **системное имя** с **идентификатором пользователя** в системе — **UID** (User ID). **UID** — это положительное целое число, по которому система и отслеживает пользователей[1]. Обычно это чис-

[1] Это может оказаться важным, например, в такой ситуации: учётную запись пользователя с именем test удалили из системы, а потом добавили снова. Однако с точки зрения системы это уже другой пользователь, потому что у него другой UID.

ло выбирается автоматически при регистрации учётной записи, однако оно не может быть совершенно произвольным. В Linux есть некоторые соглашения относительно того, каким типам пользователей могут быть выданы идентификаторы из того или иного диапазона. В частности, **UID** от «0» до «100» зарезервированы для **псевдопользователей**[1].

Идентификатор группы (GID)

Кроме идентификационного номера пользователя с учётной записью связан **идентификатор группы**. **Группы пользователей** применяются для организации доступа нескольких пользователей к некоторым ресурсам. У группы, так же, как и у пользователя, есть имя и идентификационный номер — **GID** (Group ID). В Linux каждый пользователь должен принадлежать как минимум к одной группе — **группе по умолчанию**. При создании учётной записи пользователя обычно создаётся и группа, имя которой совпадает с **системным именем**[2], именно эта группа будет использоваться как группа по умолчанию для этого пользователя. Пользователь может входить более чем в одну группу, но в учётной записи указывается только номер группы по умолчанию. Группы позволяют регулировать доступ нескольких пользователей к различным ресурсам.

Полное имя (full name)

Помимо **системного имени** в учётной записи содержится и **полное имя** (имя и фамилия) использующего данную учётную запись человека. Конечно, пользователь может указать что угодно в качестве своего имени и фамилии. Полное имя необходимо не столько системе, сколько людям — чтобы иметь возможность определить, кому принадлежит учётная запись.

Домашний каталог (home directory)

Файлы всех пользователей в Linux хранятся раздельно, у каждого пользователя есть собственный **домашний каталог**, в котором он может хранить свои данные. Доступ других пользователей к домашнему

[1]Обычно Linux выдаёт нормальным пользователям UID, начиная с «500» или «1000».

[2]Как правило, численное значение GID в этом случае совпадает со значением UID.

каталогу пользователя может быть ограничен. Информация о домашнем каталоге обязательно должна присутствовать в учётной записи, потому что именно с него начинает работу пользователь, зарегистрировавшийся в системе.

Начальная оболочка (login shell)

Важнейший способ взаимодействовать с системой Linux — **командная строка**, которая позволяет пользователю вести «диалог» с системой: передавать ей команды и получать её ответы. Для этой цели служит специальная программа — **командная оболочка** (или **интерпретатор командной строки**), по-английски — shell. Начальная оболочка (login shell) запускается при входе пользователя в систему в текстовом режиме (например, на виртуальной консоли). Поскольку в Linux доступно несколько разных командных оболочек, в учётной записи указано, какую из командных оболочек нужно запустить для данного пользователя. Если специально не указывать начальную оболочку при создании учётной записи, она будет назначена по умолчанию, вероятнее всего это будет bash.

Все перечисленные данные об учётных записях хранятся в файле `/etc/passwd`. Сведения о конкретной учётной записи пользователя можно получить с помощью утилиты `getent`[1]:

```
[tester@tacit tester]$ getent passwd tester
tester:x:506:506:Test Testovitch:/home/tester:/bin/bash
[tester@tacit tester]$
```

Первый параметр, `passwd` — это название базы, в которой нужно производить поиск, оно совпадает с именем соответствующего конфигурационного файла. Второй параметр, `tester` — это название учётной записи пользователя (системное имя). `getent` выводит ту строчку `/etc/passwd`, где описана искомая учётная запись: в ней через «:» указаны системное имя, пароль (тут стоит буква «x», потому что пароль спрятан в другом месте, об этом ниже), UID, GID, полное имя, домашний каталог и начальная оболочка.

В Linux пароль пользователя в явном виде не хранится нигде, но только в зашифрованном. В современных системах обычно применяются так называемые «теневые пароли» (shadow passwords), которые хранятся отдельно от остальных сведений об учётной записи, а также

[1]Эта утилита также полезна для получения сведений о некоторых других системных ресурсах, см. `getent --help`.

позволяют назначать дополнительные ограничения, в частности, «срок годности» пароля. В зависимости от строгости политики безопасности зашифрованные пароли пользователей могут храниться в общем файле `/etc/shadow` (менее строго) или в отдельном файле `shadow` для каждого пользователя. В ALT Linux по умолчанию используется схема `tcb`, реализующая более строгую политику. Для просмотра сведений из файла `shadow` требуются полномочия суперпользователя, это можно сделать с помощью команды `getent passwd tester`. Подробнее о возможностях теневых паролей можно прочитать в руководствах `shadow(5)` и `tcb(5)`.

Также отдельно хранится информация обо всех группах пользователей в системе, для этого предназначен файл `/etc/group`. Информацию о конкретной группе можно получить с помощью той же утилиты `getent`:

```
[tester@tacit tester]$ getent group audio
audio:x:81:yura,fedya
[tester@tacit tester]$
```

Запись в файле `/etc/group` устроена очень просто: сначала идёт имя группы (как и имя учётной записи, потом поле для пароля (здесь опять «x», но пароли для группы используются очень редко), GID, список через запятую названий учётных записей (имён пользователей), входящих в данную группу. Любой пользователь может получить список названий групп, в которых он состоит командой `groups`, а более подробные сведения о своей или чужой учётной записи командой `id имя_пользователя`. Принадлежность к группе существенна только в одном отношении — прав доступа, поскольку для каждого файла определён не только пользователь-владелец, но и группа-владелец.

Управление пользователями

Создание пользователей

Для создания полноценного пользователя Linux нужно выполнить несколько относительно независимых действий:

- создать запись в `/etc/passwd`, где присвоить учётной записи уникальное имя, UID и пр.;
- создать домашний каталог пользователя, обеспечить пользователю доступ к его домашнему каталогу (сделать его владельцем каталога);

- поместить в домашний каталог стандартное наполнение (обычно конфигурационные файлы), взятое из `/etc/skel`;
- модифицировать системные конфигурационные файлы, в частности, создать хранилище для приходящей почты для данного пользователя (`/var/spool/mail/tester`).

Все эти действия могут быть выполнены и вручную, однако это довольно неудобно и можно что-нибудь забыть. Для упрощения процесса используется утилита `useradd` (она же по традиции называется `adduser`), для выполнения которой, естественно, потребуются полномочия администратора. В простейшем случае достаточно будет двух шагов:

```
root@tacit ~# useradd test
root@tacit ~# passwd test
passwd: updating all authentication tokens for user test.

You can now choose the new password or passphrase.

. . .

Enter new password:
```

Сначала `useradd` добавляет учётную запись (имя пользователя — единственный параметр, в нашем примере — `test`), заполняя её значениями по умолчанию и проводя все необходимые изменения в системе. С помощью дополнительных параметров при вызове `useradd` можно явно указать значение для того или иного поля учётной записи, также эта утилита позволяет модифицировать параметры создания пользователей по умолчанию. Подробности можно найти в руководстве `useradd(8)`. Утилита `passwd`, вызванная с правами суперпользователя, позволяет назначить данному пользователю любой пароль. При этом сведения о предшествующем пароле данного пользователя (если таковой был), будут полностью утрачены. `passwd`, вызванная обычным пользователем (без параметров), позволяет ему сменить свой собственный пароль, но для этого потребуется ввести текущий пароль пользователя.

Существуют аналогичные `useradd` утилиты для модификации параметров уже существующей учётной записи (`usermod`) и для удаления пользователей (`userdel`). Пользователь также может изменить некоторые некритичные сведения в своей учётной записи самостоятельно.

В частности, для установки своего **полного имени** и некоторых других информационных полей учётной записи служит утилита `chfn(1)` из пакета shadow-change, сменить **начальную оболочку** поможет утилита `chsh(1)` (позволяет выбрать только одну из оболочек, перечисленных в `/etc/shells`) из того же пакета. Обратите внимание, что в ALT Linux пользователь имеет право редактировать собственную учётную запись только в том случае, если установлен соответствующий режим доступа: команда `control chsh` должна возвращать `public`, аналогично `control chfn`. Установить нужный доступ может суперпользователь командой `control chsh public` (аналогично для `chfn`).

Самую востребованную операцию по работе с группами — добавление пользователя в группу — проще всего выполнить простым редактированием файла `/etc/group`. Достаточно открыть этот файл в любом текстовом редакторе (естественно, с правами суперпользователя), найти строчку, начинающуюся с названия нужной группы, и добавить в конец этой строки имя нужного пользователя через запятую). Для управления группами существует комплект утилит `groupadd(8)`, `groupdel(8)`, `groupmod(8)`, подробности о работе с ними можно найти в соответствующих руководствах.

См. также
Alterator-users: графическое средство управления пользователями . . . . . [снаружи, стр. 47]
О ссылках на руководства вида `man(1)` (справочная система `man`) . . . . . [стр. 9]

Права доступа в системе Linux

Вадим Виниченко, Мэтт Уэлш (Matt Welsh)

Пользователи и группы

Поскольку система Linux с самого начала разрабатывалась как многопользовательская, в ней предусмотрен такой механизм, как права доступа к файлам и каталогам. Он позволяет разграничить полномочия пользователей, работающих в системе. В частности, права доступа позволяют отдельным пользователям иметь «личные» файлы

и каталоги. Например, если пользователь[1] `ivanov` создал в своём домашнем каталоге файлы, то он является владельцем этих файлов и может определить права доступа к ним для себя и остальных пользователей. Он может, например, полностью закрыть доступ к своим файлам для остальных пользователей, или разрешить им читать свои файлы, запретив изменять и исполнять их.

Правильная настройка прав доступа позволяет повысить надёжность системы, защитив от изменения или удаления важные системные файлы. Наконец, поскольку внешние устройства с точки зрения Linux также являются объектами файловой системы, механизм прав доступа можно применять и для управления доступом к устройствам.

«Пользователями» системы Linux, выполняющими различные действия с файлами и каталогами, являются на самом деле вовсе не люди, а программы, выполняемые в системе — **процессы**. Одна из таких программ — командная оболочка, которая считывает команды пользователя из командной строки и передаёт их системе на выполнение. Каждая программа (процесс) выполняется от имени определённого пользователя. Её возможности работы с файлами и каталогами определяются правами доступа, заданными для этого пользователя.

С целью оптимальной настройки прав доступа для ряда программ-серверов в системе созданы системные пользователи (учётные записи), от имени которых работают эти программы. Например, в системе ALT Linux веб-сервер (Apache) выполняется от имени пользователя apache, а ftp-сервер — от имени пользователя `ftp`. Такие учётные записи не предназначены для работы людей-пользователей.

У любого файла в системе есть **владелец** — один из пользователей. Однако каждый файл одновременно принадлежит и некоторой **группе** пользователей системы. Каждый пользователь может входить в любое количество групп, и в каждую группу может входить любое количество пользователей из числа определённых в системе.

Когда в системе создаётся новый пользователь, он добавляется по крайней мере в одну группу. В системе Linux при создании новой учётной записи создаётся специальная группа, имя которой совпада-

[1] В этом разделе, если специально не оговорено иное, под пользователем понимается пользователь с точки зрения системы, т. е. зарегистрировавшийся в определённой учётной записи (работающий под определённым именем пользователя). Права доступа определяются именно для пользователей в указанном смысле. При этом один человек, работающий в системе, может регистрироваться под различными именами пользователя для выполнения различных действий. Наоборот, несколько человек, использующих одну и ту же учётную запись, для системы являются одним и тем же пользователем.

ет с именем нового пользователя, и пользователь включается в эту группу. В дальнейшем администратор может добавить пользователя к другим группам.

Механизм групп может применяться для организации совместного доступа нескольких пользователей к определённым ресурсам. Например, на сервере организации для каждого проекта может быть создана отдельная группа, в которую войдут учётные записи (имена пользователей) сотрудников, работающих над этим проектом. При этом файлы, относящиеся к проекту, могут принадлежать этой группе и быть доступными для её членов. В системе также определено несколько групп (например, bin), которые используются для управления доступом системных программ к различным ресурсам. Как правило, членами этих групп являются системные пользователи, пользователи-люди не включаются в такие группы.

В некоторых дистрибутивах Linux (в т. ч. в дистрибутивах ALT Linux) с помощью групп могут быть предоставлены права, необходимые для выполнения определённых пользовательских задач. Например, чтобы пользователь получил возможность собирать пакеты RPM, его следует включить в группу rpm, чтобы предоставить возможность записи дисков CD-R/RW, пользователя нужно включить в группу `cdwriter` и т. д.

Виды прав доступа

Права доступа определяются по отношению к трём типам действий: чтение, запись и исполнение. Эти права доступа могут быть предоставлены трём классам пользователей: владельцу файла (пользователю), группе, которой принадлежит файл, а также всем остальным пользователям, не входящим в эту группу. Право на чтение даёт пользователю возможность читать содержимое файла или, если такой доступ разрешён к каталогам, просматривать содержимое каталога (используя команду `ls`). Право на запись даёт пользователю возможность записывать или изменять файл, а право на запись для каталога — возможность создавать новые файлы или удалять файлы из этого каталога. Наконец, право на исполнение позволяет пользователю запускать файл как программу или сценарий командной оболочки (разумеется, это действие имеет смысл лишь в том случае, если файл является программой или сценарием). Для каталогов право на исполнение имеет особый смысл — оно позволяет сделать данный каталог **текущим**, т. е. «перейти» в него, например, командой `cd`.

Чтобы получить информацию о правах доступа, используйте команду `ls` с ключом `-l`. При этом будет выведена подробная информация о файлах и каталогах, в которой будут, среди прочего, отражены права доступа. Рассмотрим следующий пример:

```
/home/ivanov/docs# ls -l report1303
-rw-r--r-- 1 ivanov users 505 Mar 13 19:05 report1303
```

Первое поле в этой строке («`-rw-r--r--`») отражает права доступа к файлу. Третье поле указывает на владельца файла («`ivanov`»), четвёртое поле указывает на группу, которая владеет этим файлом («`users`»). Последнее поле — это имя файла («`report1303`»). Другие поля описаны в документации к команде `ls`.

Данный файл является собственностью пользователя ivanov и группы users. Последовательность «`-rw-r--r--`» показывает права доступа для пользователя — владельца файла, пользователей — членов группы-владельца, а также для всех остальных пользователей.

Первый символ из этого ряда («`-`») обозначает тип файла. Символ «`-`» означает, что это — обычный файл, который не является каталогом (в этом случае первым символом было бы «`d`») или псевдофайлом устройства (было бы «`c`» или «`b`»). Следующие три символа («`rw-`») представляют собой права доступа, предоставленные владельцу ivanov. Символ «`r`» — сокращение от read (англ. читать), а «`w`» — сокращение от write (англ. писать). Таким образом, ivanov имеет право на чтение и запись (изменение) файла `report1303`.

После символа «`w`» мог бы стоять символ «`x`», означающий наличие прав на исполнение (англ. execute, исполнять) файла. Однако символ «`-`», стоящий здесь вместо «`x`», указывает, что ivanov не имеет права на исполнение этого файла. Это разумно, так как файл `report1303` не является программой. В то же время, пользователь, зарегистрировавшийся в системе как ivanov, при желании может предоставить себе право на исполнение данного файла, поскольку является его владельцем. Для изменения прав доступа к файлу или каталогу используется команда `chmod`.

Следующие три символа («`r--`») отражают права доступа группы к файлу. Группой-собственником файла в нашем примере является группа users. Поскольку здесь присутствует только символ «`r`», все пользователи из группы users могут читать этот файл, но не могут изменять или исполнять его.

Наконец, последние три символа (это опять «`r--`») показывают права доступа к этому файлу всех других пользователей, помимо соб-

ственника файла и пользователей из группы users. Так как здесь указан только символ «r», эти пользователи тоже могут читать файл

Вот ещё несколько примеров:

«-rwxr-x--x»

Пользователь-владелец файла может читать файл, изменять и исполнять его; пользователи, члены группы-владельца могут читать и исполнять файл, но не изменять его; все остальные пользователи могут лишь запускать файл на выполнение.

«-rw-------»

Только владелец файла может читать и изменять его.

«-rwxrwxrwx»

Все пользователи могут читать файл, изменять его и запускать на выполнение.

«----------»

Никто, включая самого владельца файла, не имеет прав на его чтение, запись или выполнение[1]. Хотя такая ситуация вряд ли имеет практический смысл, с точки зрения системы она является вполне корректной. Разумеется, владелец файла может в любой момент изменить права доступа к нему.

Возможность доступа к файлу зависит также от прав доступа к каталогу, в котором находится файл. Например, даже если права доступа к файлу установлены как «-rwxrwxrwx», другие пользователи не могут получить доступ к файлу, пока они не имеют прав на *исполнение* для каталога, в котором находится файл. Другими словами, чтобы воспользоваться имеющимися у вас правами доступа к файлу, вы должны иметь право на исполнение для всех каталогов вдоль пути к файлу.

Права доступа и администрирование системы

Установка и поддержание оптимальных прав доступа является одной из важнейших задач системного администратора. Права должны быть достаточными для нормальной работы пользователей и программ, но не большими, чем необходимо для такой работы. Дистрибутивы ALT Linux обладают продуманной системой прав (предо-

[1] Строго говоря, за исключением суперпользователя (root), который может выполнять любые операции над любыми файлами в системе.

пределённые группы, псевдопользователи для различных программ-серверов, права доступа для системных файлов и каталогов). Прежде чем вносить существенные изменения в эту систему, целесообразно понять её логику и выяснить, нет ли другого способа достичь нужной цели.

Поскольку программы, исполняемые от имени суперпользователя (root), могут совершать любые действия с любыми файлами и каталогами, их выполнение может нанести системе серьёзный ущерб. Это может быть как следствием уязвимостей или ошибок в программах, так и результатом ошибочных действий самого пользователя. Поэтому работа с правами суперпользователя требует особой осторожности. Чтобы уменьшить связанные с этим риски, разработчики дистрибутивов ALT Linux рекомендуют для выполнения задач, требующих таких прав, использовать утилиту sudo.

Основные команды

Ниже перечислены важнейшие команды для решения задач, связанных с правами доступа. Для получения более подробной информации об этих командах обращайтесь к руководствам по ним.

chmod

Изменение прав доступа к файлу или каталогу.

chown

Изменение владельца файла.

chgroup

Изменение группы, которой принадлежит файл.

umask

Определение прав доступа по умолчанию для файлов, создаваемых пользователем.

См. также Другие команды для работы с файловой системой [стр. 28]

Глава 4

Процессы

Процессы и управление заданиями

Мэтт Уэлш (Matt Welsh) и другие

Задания и процессы

Всякая выполняющаяся в Linux программа называется **процессом**. Linux как многозадачная система характеризуется тем, что одновременно может выполняться множество процессов, принадлежащих одному или нескольким пользователям. Вывести список исполняющихся в текущее время процессов можно командой `ps`, например, следующим образом:

```
/home/larry# ps
PID TT STAT  TIME COMMAND
24  3  S     0:03 (bash)
161 3  R     0:00 ps
/home/larry#
```

Обратите внимание, что по умолчанию команда `ps` выводит список только тех процессов, которые принадлежат запустившему её пользователю. Чтобы посмотреть все исполняющиеся в системе процессы, нужно подать команду `ps -a`. **Номера процессов** (process ID, или PID), указанные в первой колонке, являются уникальными номерами, которые система присваивает каждому работающему процессу. Последняя колонка, озаглавленная `COMMAND`, указывает имя работающей команды. В данном случае в списке указаны процессы, которые запустил сам пользователь larry. В системе работает ещё много других процессов, их полный список можно просмотреть командой `ps -aux`. Однако среди команд, запущенных пользователем larry, есть только

bash (командная оболочка для пользователя larry) и сама команда `ps`. Видно, что оболочка bash работает одновременно с командой `ps`. Когда пользователь ввёл команду `ps`, оболочка bash начала её исполнять. После того, как команда `ps` закончила свою работу (таблица процессов выведена на экран), управление возвращается процессу bash. Тогда оболочка bash выводит на экран приглашение и ждёт новой команды.

Работающий процесс также называют **заданием** (job). Понятия процесс и задание являются взаимозаменяемыми. Однако, обычно процесс называют заданием, когда имеют ввиду **управление заданием** (job control). Управление заданием — это функция командной оболочки, которая предоставляет пользователю возможность переключаться между несколькими заданиями.

В большинстве случаев пользователи запускают только одно задание — это будет та команда, которую они ввели последней в командной оболочке. Однако многие командные оболочки (включая bash и tcsh) имеют функции **управления заданиями** (job control), позволяющие запускать одновременно несколько команд или **заданий** (jobs) и, по мере надобности, переключаться между ними.

Управление заданиями может быть полезно, если, например, вы редактируете большой текстовый файл и хотите временно прервать редактирование, чтобы сделать какую-нибудь другую операцию. С помощью функций управления заданиями можно временно покинуть редактор, вернуться к приглашению командной оболочки и выполнить какие-либо другие действия. Когда они будут сделаны, можно вернуться обратно к работе с редактором и обнаружить его в том же состоянии, в котором он был покинут. У функций управления заданиями есть ещё много полезных применений.

Передний план и фоновый режим

Задания могут быть либо на **переднем плане** (foreground), либо **фоновыми** (background). На переднем плане в любой момент времени может быть только одно задание. Задание на переднем плане — это то задание, с которым вы взаимодействуете; оно получает ввод с клавиатуры и посылает вывод на экран (если, разумеется, вы не перенаправили ввод или вывод куда-либо ещё). Напротив, фоновые задания не получают ввода с терминала; как правило, такие задания не нуждаются во взаимодействии с пользователем.

Некоторые задания исполняются очень долго, и во время их работы не происходит ничего интересного. Пример таких заданий — компили-

рование программ, а также сжатие больших файлов. Нет никаких причин смотреть на экран и ждать, когда эти задания выполнятся. Такие задания следует запускать в фоновом режиме. В это время вы можете работать с другими программами.

Для управления выполнением процессов в Linux предусмотрен механизм передачи **сигналов**. Сигнал — это способность процессов обмениваться стандартными короткими сообщениями непосредственно с помощью системы. Сообщение-сигнал не содержит никакой информации, кроме номера сигнала (для удобства вместо номера можно использовать предопределённое системой имя). Для того, чтобы передать сигнал, процессу достаточно задействовать системный вызов `kill()`, а для того, чтобы принять сигнал, не нужно ничего. Если процессу нужно как-то по-особенному реагировать на сигнал, он может зарегистрировать **обработчик**, а если обработчика нет, за него отреагирует система. Как правило, это приводит к немедленному завершению процесса, получившего сигнал. Обработчик сигнала запускается *асинхронно*, немедленно после получения сигнала, что бы процесс в это время ни делал.

Два сигнала — номер 9 (`KILL`) и 19 (`STOP`) — всегда обрабатывает система. Первый из них нужен для того, чтобы убить процесс наверняка (отсюда и название). Сигнал `STOP` *приостанавливает* процесс: в таком состоянии процесс не удаляется из таблицы процессов, но и не выполняется до тех пор, пока не получит сигнал 18 (`CONT`) — после чего продолжит работу. В командной оболочке Linux сигнал `STOP` можно передать активному процессу с помощью управляющей последовательности *Ctrl–Z*.

Сигнал номер 15 (`TERM`) служит для прерывания работы задания. При **прерывании** (interrupt) задания процесс погибает. Прерывание заданий обычно осуществляется управляющей последовательностью *Ctrl–C*[1]. Восстановить прерванное задание никаким образом невозможно. Следует также знать, что некоторые программы перехватывают сигнал `TERM` (при помощи обработчика), так что нажатие комбинации клавиш *Ctrl–C* (о) может не прервать процесс немедленно. Это сделано для того, чтобы программа могла уничтожить следы своей работы прежде, чем она будет завершена. На практике, некоторые программы вообще нельзя прервать таким способом.

[1]Прерывающая комбинация клавиш может быть установлена с помощью команды `stty`.

Перевод в фоновый режим и уничтожение заданий

Начнём с простого примера. Рассмотрим команду yes, которая на первый взгляд может показаться бесполезной. Эта команда посылает бесконечный поток строк, состоящих из символа «y» на стандартный вывод. Посмотрим, как работает эта команда:

```
/home/larry# yes
y
y
y
y
y
```

Последовательность таких строк будет бесконечно продолжаться. Уничтожить этот процесс можно, отправив ему сигнал прерывания, т. е. нажав *Ctrl–C*. Поступим теперь иначе. Чтобы на экран не выводилась эта бесконечная последовательность перенаправим стандартный вывод команды yes на `/dev/null`. Как вы, возможно, знаете, устройство `/dev/null` действует как «чёрная дыра»: все данные, посланные в это устройство, пропадают. С помощью этого устройства очень удобно избавляться от слишком обильного вывода некоторых программ.

```
/home/larry# yes > /dev/null
```

Теперь на экран ничего не выводится. Однако и приглашение командной оболочки также не возвращается. Это происходит потому, что команда yes все ещё работает и посылает свои сообщения, состоящие из букв «y» на `/dev/null`. Уничтожить это задание также можно, отправив ему сигнал прерывания.

Допустим теперь, что вы хотите, чтобы команда yes продолжала работать, но при этом и приглашение командной оболочки должно вернуться на экран, так чтобы вы могли работать с другими программами. Для этого можно команду yes перевести в фоновый режим, и она будет там работать, не общаясь с вами.

Один способ перевести процесс в фоновый режим — приписать символ «&» к концу команды. Пример:

```
/home/larry# yes > /dev/null &
[1]+ 164
/home/larry#
```

Сообщение «[1]» представляет собой **номер задания** (job number) для процесса yes. Командная оболочка присваивает номер задания каждому исполняемому заданию. Поскольку yes является единственным исполняемым заданием, ему присваивается номер 1. Число «164» является идентификационным номером, соответствующим данному процессу (PID), и этот номер также дан процессу системой. Как мы увидим дальше, к процессу можно обращаться, указывая оба этих номера.

Итак, теперь у нас есть процесс команды `yes`, работающий в фоне, и непрерывно посылающий поток из букв у на устройство `/dev/null`. Для того, чтобы узнать статус этого процесса, нужно исполнить команду `jobs`, которая является внутренней командой оболочки.

```
/home/larry# jobs
[1]+  Running                 yes >/dev/null  &
/home/larry#
```

Мы видим, что эта программа действительно работает. Для того, чтобы узнать статус задания, можно также воспользоваться командой `ps`, как это было показано выше.

Для того, чтобы передать процессу сигнал (чаще всего возникает потребность *прервать* работу задания) используется утилита `kill`. В качестве аргумента этой команде даётся либо номер задания, либо PID. Необязательный параметр — номер сигнала, который нужно отправить процессу. По умолчанию отправляется сигнал `TERM`. В рассмотренном выше случае номер задания был 1, так что команда `kill %1` прервёт работу задания. Когда к заданию обращаются по его номеру (а не PID), тогда перед этим номером в командной строке нужно поставить символ процента («%»).

Теперь введём команду `jobs` снова, чтобы проверить результат предыдущего действия:

```
/home/larry# jobs [1]
Terminated              yes      >/dev/null
```

Фактически задание уничтожено, и при вводе команды jobs следующий раз на экране о нем не будет никакой информации.

Уничтожить задание можно также, используя идентификационный номер процесса (PID). Этот номер, наряду с идентификационным номером задания, указывается во время старта задания. В нашем примере значение PID было 164, так что команда `kill 164` была бы эквивалентна команде `kill %1`. При использовании PID в качестве аргумента команды `kill` вводить символ «%» не требуется.

Приостановка и продолжение работы заданий

Запустим сначала процесс командой yes на переднем плане, как это делалось раньше:

```
/home/larry# yes > /dev/null
```

Как и ранее, поскольку процесс работает на переднем плане, приглашение командной оболочки на экран не возвращается.

Теперь вместо того, чтобы прервать задание комбинацией клавиш *Ctrl–C*, задание можно **приостановить** (suspend, буквально — «подвесить»), отправив ему сигнал `STOP`. Для приостановки задания надо нажать соответствующую комбинацию клавиш, обычно это *Ctrl–Z*.

```
/home/larry# yes > /dev/null
CtrlZ[1]+  Stopped yes       >/dev/null
/home/larry#
```

Приостановленный процесс попросту не выполняется. На него не тратятся вычислительные ресурсы процессора. Приостановленное задание можно запустить выполняться с той же точки, как будто бы оно и не было приостановлено.

Для возобновления выполнения задания на переднем плане можно использовать команду `fg` (от слова «foreground» — передний план).

```
/home/larry# fg
yes >/dev/null
```

Командная оболочка ещё раз выведет на экран название команды, так что пользователь будет знать, какое именно задание он в данный момент запустил на переднем плане. Приостановим это задание ещё раз нажатием клавиш *Ctrl–Z*, но в этот раз запустим его в фоновый режим командой `bg` (от слова background — фон). Это приведёт к тому, что данный процесс будет работать так, как если бы при его запуске использовалась команда с символом «&» в конце (как это делалось в предыдущем разделе):

```
/home/larry# bg
[1]+ yes $>$/dev/null  &
/home/larry#
```

При этом приглашение командной оболочки возвращается. Сейчас команда `jobs` должна показывать, что процесс `yes` действительно в данный момент работает; этот процесс можно уничтожить командой `kill`, как это делалось раньше.

Для того, чтобы приостановить задание, работающее в фоновом режиме, нельзя воспользоваться комбинацией клавиш *Ctrl–Z*. Прежде, чем приостанавливать задание, его нужно перевести на передний план командой `fg` и лишь потом приостановить. Таким образом, команду `fg` можно применять либо к приостановленным заданиям, либо к заданию, работающему в фоновом режиме.

Между заданиями в фоновом режиме и приостановленными заданиями есть большая разница. Приостановленное задание не работает — на него не тратятся вычислительные мощности процессора. Это задание не выполняет никаких действий. Приостановленное задание занимает некоторый объем оперативной памяти компьютера, через некоторое время ядро откачает эту часть памяти на жёсткий диск «до востребования». Напротив, задание в фоновом режиме выполняется, использует память и совершает некоторые действия, которые, возможно, вам требуются, но вы в это время можете работать с другими программами.

Задания, работающие в фоновом режиме, могут пытаться выводить некоторый текст на экран. Это будет мешать работать над другими задачами.

```
/home/larry# yes &
```

Здесь стандартный вывод не был перенаправлен на устройство `/dev/null`, поэтому на экран будет выводится бесконечный поток символов «y». Этот поток невозможно будет остановить, поскольку комбинация клавиш *Ctrl–C* не воздействует на задания в фоновом режиме. Для того чтобы остановить эту выдачу, надо использовать команду `fg`, которая переведёт задание на передний план, а затем уничтожить задание комбинацией клавиш *Ctrl–C*.

Сделаем ещё одно замечание. Обычно командой `fg` и командой `bg` воздействуют на те задания, которые были приостановлены последними (эти задания будут помечены символом «+» рядом с номером задания, если ввести команду `jobs`). Если в одно и то же время работает одно или несколько заданий, задания можно помещать на передний план или в фоновый режим, задавая в качестве аргументов команды `fg` или команды `bg` их идентификационный номер (job ID). Например,

команда `fg %2` помещает задание номер 2 на передний план, а команда `bg %3` помещает задание номер 3 в фоновый режим. Использовать PID в качестве аргументов команд `fg` и `bg` нельзя.

Более того, для перевода задания на передний план можно просто указать его номер. Так, команда `%2` будет эквивалентна команде `fg %2`.

Важно помнить, что функция управления заданием принадлежит оболочке. Команды `fg`, `bg` и `jobs` являются внутренними командами оболочки. Если, по некоторой причине, вы используете командную оболочку, которая не поддерживает функции управления заданиями, то вы в ней этих (и подобных) команд не отыщете.

См. также О перенаправлении ввода и вывода в командной оболочке . . . . . [стр. 112]

Что происходит в системе?

Георгий Курячий

Человеку, отвечающему за работоспособность системы, очень важно всегда отчётливо представлять, что с нею творится. Теоретически, никакое происшествие не должно ускользнуть от его внимания. Однако компьютерные системы столь сложны, что отслеживать *все* события в них — выше человеческих возможностей. Для того, чтобы довести поток служебной информации до разумного объёма, её надо **просеять** (выкинуть незначащие данные), **классифицировать** (разделить на несколько групп сообразно тематике) и **журнализировать** (сохранить в доступном виде для дальнейшего анализа).

В Linux эта задача решается с помощью механизма **централизованной журнализации**, который реализован демоном (системной службой) syslogd. Все части системы (включая ядро и системные службы) рапортуют syslogd о происходящих в них событиях. В этот рапорт включается имя службы, **категория** (facility) и **важность** (priority) произошедшего события. Демон, сообразно настройкам, классифицирует все эти рапорты в несколько выходных потоков. Классификация и отсев данных всякого выходного потока происходит так: для каждой категории событий определяется *наименьшая* важность, которой событие должно обладать, чтобы попасть в этот выходной поток. Например, легко определить поток «ошибки», в который будут попадать только *важные* рапорты *любых* категорий, или поток «безопасность», в который будут попадать *все* рапорты катего-

рии «безопасность» и те рапорты других категорий, важность которых заставляет подозревать угрозу безопасности системы (например, рапорт категории «демон» об аварийном завершении работы системной службы).

Главное место хранения уже классифицированного syslogd потока событий — **системный журнал** (т. н. log-файл). Системный журнал — *текстовый* файл, содержащий рапорты *одного* потока. Обычно syslogd хранит системные журналы в каталоге `/var/log` и его подкаталогах. Именно в системные журналы, прежде всего в `/var/log/messages`, `/var/log/maillog` и `/var/log/dmesg`, необходимо заглядывать администратору, который хочет знать, что происходит в системе. Поток рапортов о *важных* событиях syslogd направляет и на **системную консоль** — выделенное терминальное устройство. В ALT Linux роль системной консоли выполняет 12-я *виртуальная консоль*, доступная по сочетанию клавиш *Alt–F12* или *Alt–Ctrl–F12*. Стоит заметить, что некоторые службы (например, WWW-сервер apache) *самостоятельно*, в обход syslogd, ведут журнализацию своих событий, поэтому информацию о количестве и местоположении их журналов можно почерпнуть из их файлов настроек (обычно, тем не менее, журналы хранятся в `/var/log`).

Новые рапорты, поступающие в системный журнал, наиболее актуальны, а предыдущие, по мере их устаревания, эту актуальность утрачивают. Если самые старые данные в журнале не удалять, файловая система, рано или поздно, окажется переполненной. В Linux организован механизм **устаревания журналов**, которым занимается служба logrotate. Запускаясь раз в день, logrotate проверяет, какие из файлов следует признать устаревшими. Файл объявляется устаревшим раз в определённый промежуток времени (например, раз в неделю), или если он достиг определённого размера.

Процедура устаревания такова. Для каждого журнала, как, например, для `/var/log/syslog/alert`, logrotate держит в том же каталоге *очередь устаревших копий* — файлы с именами от `alert.0.bz2` (предыдущая копия) до `alert.4.bz2` (самая старая копия). Очередь `alert` в нашем примере состоит из пяти упакованных с помощью bzip2 файлов. В момент устаревания `alert.3.bz2` переименовывается в `alert.4.bz2` (старые данные теряются), копия с номером «2» превращается в третью, первая — во вторую, нулевая в первую. Наконец, сам журнал упаковывается и переименовывается в `alert.0.bz2`, а на его месте заводится новый — пустой. Таким образом, администратор всегда имеет

доступ к *свежему* журналу и к нескольким его копиям за определённое время.

Некоторые файлы в `/var/log` — нетекстовые, они не являются полноценными журналами, а представляют собой «свалку событий» для служб авторизации и учёта. Текстовую информацию о входе пользователей в систему и выходе оттуда можно получить по команде `last`, а узнать о тех, кто в данный момент пользуется системой, помогут команды `w` и `who`.

Множество важной информации может дать анализ *загруженности* системы — в плане процессорного времени и потребления оперативной памяти (`ps`, `top`, `vmstat`), в плане использования дискового пространства (`du`, `df`, `lsof`) и в плане работы сетевых устройств (`netstat`).

См. также | Системные журналы программы установки ALT Linux [снаружи, стр. 27]

Часть II

Linux общественный

Глава 5

Не бесплатное, а свободное

Программное обеспечение: право и свобода

Кирилл Маслинский, Алексей Смирнов

Частное и общественное

Написание компьютерных программ — не так уж давно возникшая форма интеллектуальной деятельности. В написании программы действительно много общего с написанием какого-нибудь литературного или другого нетривиального текста, поэтому совершенно естественно, что с точки зрения правовых отношений программы попали в один класс с такими текстами — «произведений».

Эти правовые отношения регулируются законодательством об авторском праве и на сегодняшний день тесно ассоциируются с широко распространённой системой экономического использования этих прав: торговля экземплярами произведения и запрет на тиражирование произведения (создание новых экземпляров) для всех, кроме обладателя прав. В этой модели распространения к произведению относятся как к собственности правообладателя. Программы, которые распространяются по такой модели, называются точным, но не очень благозвучным в русской огласовке термином **проприетарные**.

Однако не менее широко распространена и другая модель, в которой к произведению относятся как к общественному достоянию, плоду интеллектуального творчества, который должен быть доступен любо-

му и не принадлежать никому в отдельности[1]. Типичный пример — произведения давно умерших классиков, хотя и многие здравствующие авторы (например, учёные) распространяют свои произведения по этой, общественной модели. Несмотря на молодость, программное обеспечение тоже может следовать общественной модели распространения, и здесь возникают два ключевых понятия: **свободное ПО** и **открытые системы**.

Заметьте, что речь не идёт о платном и бесплатном. Экземпляры произведений классиков тоже продаются за деньги, разница в том, что правомерно издавать (тиражировать) произведение, находящееся в общественном достоянии, может любой.

У программ для компьютера есть всё-таки несколько существенных свойств, которые отличают их от текстов на естественном языке, поэтому и при распространении ПО возникают некоторые специфические особенности. Чтобы разобраться в смысле понятий «открытые системы» и «свободное ПО», нам потребуется сформулировать два свойства компьютерных программ:

1) Очень часто, хотя и не обязательно, программа существует в двух видах: производится она в одной форме — в виде **исходного текста**, а распространяется и используется в другой — в виде скомпилированной **двоичной программы**, машинных кодов, по которым невозможно однозначно восстановить исходный текст.

2) Объектом авторского права (то есть **произведением**) является исходный текст программы. Двоичная форма — это уже экземпляр произведения, и в его отношении действуют примерно те же нормы, что и к экземпляру книги, с некоторыми технологическими оговорками. Правовая разница между произведением и экземпляром произведения, а также применение авторского права к программному обеспечению неюридическим языком хорошо разъясняется в Anti Copyright FAQ[2] Фёдора Зуева.

[1] Нужно отметить, что это относится только к имущественной стороне авторского права, неимущественные же права, в частности, право на имя, на сохранение целостности произведения и т. п., в конечном итоге, право на репутацию автора, всегда сохраняются за автором, вне зависимости от модели распространения. Такие права не могут быть проданы или переданы другому лицу.

[2] `http://www.libertarium.ru/libertarium/Anti-Copyright-FAQ`

Вообще говоря, свобода и открытость — независимые признаки, поэтому мы рассмотрим их по отдельности.

Степени открытости

Нулевая степень

То, что большинство программ используются в двоичной форме и не требуют для работы наличия исходного текста, приводит к возможности распространять двоичные экземпляры программы, никому не показывая исходные тексты. Это подкрепляется рассуждением: конечного пользователя в первую очередь интересует программа как работающий продукт, а не то, как и почему она работает. Такое программное обеспечение широко распространено на сегодняшнем рынке и может быть точно обозначено как **программное обеспечение с закрытым исходным текстом**.

Минимальная открытость: спецификация форматов и интерфейсов

Если каждую программу сделать «вещью в себе», которая работает одной ей ведомыми способами с одной ей понятными данными, то будет невозможно какое бы то ни было взаимодействие разных программ и их совместное использование. Недостатки очевидны: на каждого автора или производителя программы наваливается необходимость все задачи решать самостоятельно, не имея возможности перепоручить часть работы другой программе; невозможен будет обмен данными между пользователями, если у них нет одной и той же программы (а что будет, если разные версии программы обрабатывают данные слегка по-разному?). Поэтому естественно, что идеальная модель полностью закрытого ПО никогда не была реализована.

С другой стороны, чтобы обеспечить взаимодействие программ, совершенно не требуется целиком открывать их исходный текст. Достаточно чётко описать, каким способом можно обращаться к программе, чтобы добиться определённого результата, и в каком виде программа возвратит этот результат. Такое описание называется **спецификация интерфейса взаимодействия** или **API**.

Чтобы обеспечить обмен данными, необходимо составить набор правил, в соответствии с которыми определённые данные могут быть

переведены в форму, доступную для обработки программой, и наоборот, как из этой формы восстановить смысл закодированных в ней данных. Такой набор правил называется **спецификацией формата данных**.

В качестве примера можно привести текстовый документ. Программа представляет документ в виде некоторого файла. Если у нас нет спецификации формата этого файла, то единственный способ получить содержащиеся в нём данные — прибегнуть к помощи этой самой программы, которая неизвестным же для нас образом их извлечёт и отобразит в понятном виде. Теперь представим себе, что это не просто текстовый документ, а государственная бумага, подписанная электронно-цифровой подписью официального лица. Поскольку мы не знаем, как именно программа делает из файла читаемый документ, то даже удостоверившись с помощью электронно-цифровой подписи в подлинности *файла*, мы не можем гарантировать, что видим *тот же самый текст*, который был подписан официальным лицом. Например, на одной из сторон программа настроена иначе и не отобразила примечания. . . Катастрофические государственные последствия очевидны.

Такое невозможно, если мы располагаем строгой спецификацией формата файла, содержащего документ. Мы всегда, даже без участия какой бы то ни было специальной программы, сможем восстановить из файла содержащийся в нём текст, если нам известен полный набор правил, по которым это делается. При необходимости мы можем создать собственную программу, которая будет работать с файлами в этом формате.

Неполная открытость

Приведённый выше пример показывает, что есть случаи, когда требования пользователя программы шире, чем просто работающая программа: иногда необходимо точно знать устройство и принцип работы программы, например, чтобы иметь возможность исключить недостоверность и различное толкование данных. В такой ситуации автор или производитель вправе предоставить исходные тексты своей программы некоторому закрытому сообществу — клиенту или государственному органу — обычно на условиях неразглашения.

Тем не менее, такая условная открытость не может добавить общественного доверия, она только расширяет круг тех лиц, на авторитетном заявлении которых держится уверенность в тех или иных результатах работы программы.

Другая возможность — публичная демонстрация не полного исходного текста программы, а только отдельных его фрагментов. Принципиально такой подход не добавляет открытости и общественного доверия.

Максимальная открытость

Программное обеспечение, исходные тексты которого опубликованы или предоставляются любому по первому запросу, называется **программным обеспечением с открытым исходным текстом**, это несколько многословный аналог более лаконичного термина **Open Source Software**.

Но даже если публике представлен исходный текст программы целиком, этого ещё не достаточно, чтобы считать программу полностью открытой, а её работу — полностью прозрачной для пользователя. Для этого ещё необходима уверенность в том, что используемая программа в двоичном виде действительно была получена непосредственно из данного исходного текста. Гарантировать это можно, если не только исходный текст, но и инструментарий, с помощью которого он преобразуется в двоичный вид, будут открыты для публики. В таком случае двоичную программу можно просто воспроизвести (заново скомпилировать), что даёт возможность контроля и аудита программного обеспечения.

Смысл открытых систем

Даже минимальная открытость уже даёт независимость от конкретного производителя ПО, гарантии целостности и однозначности данных, открывает дорогу взаимодействию и совместной работе программ. Бóльшая открытость ведёт к повышению общественного доверия к программе. Если же программное обеспечение применяется в общественной и государственной сфере, его открытость является гарантией соблюдения принципов гражданского общества[1].

[1] Если и делать государственный документооборот электронным, то этот механизм должен быть полностью открытым, так как он служит ограничению прав. Это полностью аналогично тому, как должны быть полностью открыты для публики тексты законов.

Степени свободы

По российскому законодательству программное обеспечение не может быть запатентовано, поэтому все связанные с ним имущественные и неимущественные отношения в России регулируются только законодательством о программах для ЭВМ (ЗоПЭВМ) и об авторском праве и смежных правах.

Право автора подписывать произведение своим именем и прочие неимущественные авторские права всегда сохраняются за автором и не могут быть переданы другому лицу; в дальнейшем изложении эта сторона авторского права не будет обсуждаться. Имущественные же авторские права касаются разных форм тиражирования произведения, в том числе передачи в эфир, переработки и т. д. В момент создания произведения все имущественные права принадлежат исключительно автору, а всем остальным тиражирование произведения запрещается законом. С имущественными правами автор волен поступать по своему усмотрению, главным образом, передавать их любому лицу или организации. Условия передачи определяются в **авторском договоре**, и могут быть произвольными в рамках допустимых договорных обязательств.

В современном мире появилась новая форма заключения договора: одна из сторон предлагает текст договора в электронном виде, а другая, прочтя его с экрана, принимает условия, нажав на какую-нибудь кнопку. В частности, таким образом производитель или автор программы может распространять вместе со своей программой и авторский договор. Текст договора может быть и просто приложен к программе. Для обозначения такой формы договора часто используют слово «лицензия» — кальку с английского license. Однако в России юридического смысла это слово не имеет, правильно в данном случае говорить об авторском договоре.

Общественная модель распространения произведений предполагает известную степень свободы в тиражировании произведения. Законодательством об авторском праве предусмотрен срок, по прошествии которого произведение переходит в общественное достояние, фактически имущественные права на произведение в этот момент уничтожаются[1]. Однако пока, видимо, ещё ни одна компьютерная программа не достигла преклонного возраста, достаточного для перехода в общественное достояние. Тем не менее, в силах автора сделать своё произведение распространяемым по общественной модели — вопрос только в объёме имущественных прав, которые автор или законный правообладатель готовы передать в общественное достояние.

[1] На сегодняшний день по российскому законодательству этот срок весьма долог — 70 лет с момента смерти автора.

Нулевая степень свободы

Минимальная степень свободы — использовать программу любым способом и с любой целью. Для российских пользователей ПО эта свобода действительно «нулевая», в том смысле, что она присутствует всегда, что бы ни говорилось в авторском договоре. По российскому законодательству обладатель имущественных авторских прав волен ограничивать право пользователей на *тиражирование* своего произведения, однако у него нет никаких прав каким бы то ни было образом ограничивать владельца экземпляра произведения в *использовании* программы.

Нулевая свобода — это некоторая гарантия личной свободы пользователя от посягательств производителей ПО, но она ещё не является свободой для самого произведения.

Свобода распространения

Первая свобода собственно для произведения — свобода распространения. Для автора она означает, с одной стороны, снятие всяких ограничений на тиражирование произведения (и потенциально — его более широкое распространение); с другой стороны, фактический отказ от получения вознаграждения за передачу имущественных прав.

Известно огромное число примеров свободно распространяемых программ. Однако свобода распространения никак не предполагает, что должны быть доступны исходные тексты программы, вполне можно распространять программу только в двоичном виде. Зачастую именно так и происходит: производитель ПО стремится максимально широко распространить свою программу среди пользователей, даже отказываясь от платы за экземпляры, при этом не делает свою программу открытой. Отпущенная «в свободное плавание» программа ещё не может считаться вполне общественным достоянием, если её исходный текст — который и является собственно произведением — недоступен публике.

Свобода модификации

Одно из имущественных авторских прав, оговорённых в законодательстве, — право на переработку произведения. Это право приобретает очень большое значение, если в качестве произведения рассматривается компьютерная программа. В отличие от литературных и прочих текстов, программа должна *работать*. Это значит, что в ней необходимо исправлять ошибки, приспосабливать для работы в новых системах и т. п. Если исходный текст программы открыт, то в силах любого компетентного человека внести необходимые исправления. Для общественности эта возможность имеет значение, если есть свобода распространения модифицированных версий программы. Ведь автор не всегда доступен, не всегда заинтересован и имеет возможность вносить исправления или переносить программу на другие системы.

Если в авторском договоре, сопровождающем программу, автор передаёт право модифицировать и распространять модифицированные версии программы, то такую программу можно с полным правом отнести к **свободному ПО** (Free Software), в том понимании, которое было сформулировано Ричардом Столлманом[1]. Естественно, свобода модификации предполагает, что открыт исходный текст программы.

На сегодняшний день распространено довольно много типовых форм авторских договоров для распространения свободных программ. Те из них, которые удовлетворяют требованиям к свободному ПО, перечислены на сайте Фонда свободного программного обеспечения[3]. Названную свободу модификации в полной мере реализует лицензия BSD[4] [5].

Максимальная свобода

Может ли общественность злоупотребить данной ей свободой в обращении с программой? Вполне. Самое страшное злоупотребление, которое можно себе представить — модифицировав программу, запретить свободное распространение модифицированной версии и даже закрыть её исходный текст. Таким образом программа может быть

[1] Имеются в виду 4 свободы Столлмана, декларированные в манифесте созданного им Фонда свободного программного обеспечения (Free Software Foundation[2]).

[3] `http://www.gnu.org/licenses/license-list.ru.html`

[4] `http://ru.wikipedia.org/wiki/BSD_License`

[5] BSD — **B**erkeley **S**oftware **D**istribution, пакет совместимого с UNIX программного обеспечения, разработанный в университете Беркли и распространявшийся свободно.

изъята из свободного обращения и переведена в частное владение. В соответствии с лицензией BSD такое поведение является абсолютно законным и предусмотренным: объём передаваемых авторских прав достаточен для таких действий.

Однако не всегда автор, отдавая своё произведение общественности, согласится на такое его использование. Чтобы избежать «закрепощения» своей программы, автор может воспользоваться тем же самым законным инструментом — авторским договором. Главный прототип такого договора (да и вообще главный прототип всех свободных лицензий) — GPL[1], общественная лицензия GNU (GNU **G**eneral **P**ublic **L**icense), впервые сформулированная тем же Ричардом Столлманом. Эта лицензия, помимо предоставления всех необходимых свобод, включает условие **copyleft**: никто не имеет права, сделав модифицированную версию свободной программы, распространять её, не соблюдая *всех* принципов свободного ПО, ограничивая тем самым права *других пользователей* по отношению к программе. Говоря короче, запрещает модификацию свободной программы делать несвободной.

Любой авторский договор, включающий такое условие, может быть назван «copyleft». Это игра слов с умыслом: по-английски авторское право называется «copyright», буквально «копироватьправо», а «copyleft», соответственно, «копироватьлево». Действительно, условие «copyleft» прямо противоположно по смыслу авторскому праву: авторское право призвано ограничить пользователя в копировании и распространении копий продукта, а «авторское лево», наоборот, строго запрещает его ограничивать. «Авторское лево» реализует идею о том, что интеллектуальные достижения человека не могут и не должны находиться в чьей-то частной собственности, и сохраняет свободу наилучшим способом — пользуясь теми самыми механизмами ограничения, которые предоставлены законодательством об авторском праве.

Несмотря на то, что лицензия BSD с юридической точки зрения передаёт бóльший объём прав, в нашем изложении она оказалась ниже на шкале свободы именно по той причине, что она даёт меньше гарантий свободы программе.

Смысл свободного ПО

Причины, по которым люди и организации выбирают свободную модель распространения своего программного обеспечения, очень раз-

[1] `http://www.infolex.narod.ru/gpl_gnu/gplrus.html`

нообразны и индивидуальны. Одна из важнейших и самых общих причин для авторов — это стремление к свободе интеллектуальной деятельности. Яркое выражение этого стремления и опасений за интеллектуальную свободу можно найти в очень коротком антиутопическом рассказе Ричарда Столлмена «Право читать»[1]. Однако здесь не всё универсально, и в общем случае существуют две противоположные стратегии поведения по отношению к полученным интеллектуальным результатам: спрятать подальше или распространить пошире, предполагающие, соответственно, частную и общественную модели распространения. Научные и университетские традиции склоняются в пользу второй стратегии, но стоит науке подойти достаточно близко к технологии — как нередко актуализируется частная модель, возникают патенты и закрытые результаты.

Для индивидуальных авторов на некоммерческом поприще общественная модель — это хорошая возможность для самореализации: на свободно распространяемых программах всегда стоят имена их авторов; здесь же и возможности для социализации — вокруг удачных и востребованных свободных программ всегда складывается сообщество разработчиков и пользователей.

Общественная модель распространения ПО представляет хорошие возможности и для бизнеса. К началу XXI века уже стало традиционным строить бизнес в области программного обеспечения из двух компонент: собственно разработки и торговли лицензиями (правом производить ограниченное количество экземпляров), причём вторая часть значительно прибыльнее первой, поскольку расходы не увеличиваются пропорционально числу проданных лицензий. На рынке свободного ПО вторая компонента просто отпадает — производить экземпляры разрешено всем в неограниченных количествах, а первая — собственно разработка — остаётся в полном объёме. В результате преимущество получают те, кто делает реальную разработку — это открывает дорогу конкуренции и инновациям. В целом, на рынке свободного ПО значительно ниже финансовый порог вхождения при достаточно высоком интеллектуальном пороге.

Свобода и открытость Linux

Если быть точным, слово Linux обозначает только **ядро** операционной системы, хоть и самый главный, но всё же только один из ком-

[1] http://www.gnu.org/philosophy/right-to-read.ru.html

понентов сложнейшей системы, какой является операционная среда, в которой работает пользователь. Всё остальное — утилиты, графическая среда, пользовательские приложения, — это независимые друг от друга *свободные* программы, над каждой из которых работает своя команда разработчиков и пользователей. Само по себе ядро Linux тоже является совершенно независимой свободной программой. Современные дистрибутивы, включающие в себя все эти компоненты, тоже обозначают словом Linux, но это уже его расширительное понимание.

Сформировать целостную операционную систему из независимо разрабатываемых компонентов может только одно — соблюдение открытых стандартов на форматы и интерфейсы, что позволяет организовать взаимодействие программ и обмен данными. В этом отношении Linux наследует традициям открытого проектирования семейства операционных систем, обозначаемых общим и несколько собирательным именем Unix. Исторически в Unix сложился фактический стандарт на интерфейс операционной системы (**API**), который впоследствии был даже зафиксирован в виде официального стандарта под именем POSIX. Ядро Linux было написано финским студентом, Линусом Торвальдсом, как независимая и свободная реализация Unix API с открытым исходным текстом.

К моменту первой публикации Linux в 1991 году уже существовало значительное количество свободных программ для Unix, самые важные разрабатывались в рамках проекта GNU[1] Ричарда Столлмана. Однако не существовало свободного Unix-совместимого ядра. Сложилось так, что ядро Linux стало тем связующим звеном, которое позволило собрать вместе ПО, которое уже было свободным, и создать первую полностью *свободную* и *открытую* операционную систему. Это одновременно дало толчок к появлению нового свободного ПО. Поэтому сегодня свободное и открытое ПО в первую очередь ассоциируются с Linux, а Linux — со свободным ПО.

Но это совершенно не означает, что свободного и открытого нет за пределами Linux. Сам смысл свободы и открытости гарантирует то, что они никогда не окажутся неотделимы от конкретной ОС, или любого другого имени собственного. Свободное ПО пишется для самых разнообразных платформ, в том числе для Windows.

Открытость стандартов гарантирует, что инструментарий может быть перенесён на любые платформы, а следствие — многоплатфор-

[1] http://www.gnu.org/home.ru.html

менность приложений. Пример тому — проект cygwin[1], реализующий инструментарий Unix на платформе Windows.

См. также О некоторых технологиях распространения свободного ПО для Linux . . . . . [стр. 101]

Свободные программы и сообщество

Георгий Курячий

Свободное или несвободное

Когда деревья были маленькими...

Когда разработка одного экземпляра вычислительной машины длилась годами, а сам этот экземпляр выглядел как магазин продажи холодильников, программы для этой машины числились наряду с мелкими и средними деталями для неё. Скажем, для записи данных на перфоленту прежде всего необходим перфоратор, затем — процессор, который будет этим перфоратором управлять, а уж затем — всякие там провода, программы для записи данных и прочая необходимая мелочь. Красноречиво этот факт подтверждает термин «операционная система», точнее, его исходное английское написание — «operating system». Дескать, есть некоторая «система» (system) — это высокоорганизованный набор электронных деталей. Но система настолько сложна, что «использовать» (operate) эту систему лучше по определённым правилам, тут вот даже кой-какие программки для этого написаны, они и задают правила «использования системы» (operating system). О том же говорят и наши термины: «аппаратное обеспечение ЭВМ» и «программное обеспечение ЭВМ», подразумевающие, что утилиты и компиляторы так же крепко привязаны к этой вот конкретно машине, как микросхемы и вентиляторы.

Различение «набора деталей» (system) и «набора программ» (operating system) далось машиностроителям не вдруг. Сначала открыли цикл ПКМ: «программирование — компиляция — отладка (неуспешная) — программирование — и. т. д.. . . — отладка (успешная)». С чугунной деталью так не поступают: выточил деталь — поставил в машину — она взорвалась вместе с деталью... Затем выяснилось, что, в отличие от детали, программу удобно разрабатывать совместно, когда каждый может подключиться к работе и добавить несколько нужных лично ему функций; надо только определиться, кто отвечает за всю программу целиком и принимает решения.

[1] http://www.cygwin.com/

Только после распространения языка программирования «Си» и операционной системы UNIX стало зарождаться понятие **кроссплатформенности**, которое сделало это отличие совершенно очевидным, и навсегда оторвало программный продукт от жёсткой привязки к «system» и даже «operating system». Кроссплатформенность, то есть возможность использования одного и того же программного продукта независимо от того, где он запускается («платформы» или «среды»), открыла новые возможности в совместной разработке. К одному и тому же проекту стали присоединяться программисты, работающие на разных компьютерах. Они свободно обменивались друг с другом программами, но ещё чаще — исходными текстами к ним, так как выполняться скомпилированная программа может только в рамках одной платформы, а исходные тексты всё равно нужны для доработки и исправления.

Безущербное копирование и продажа воздуха

Где-то в это время, в конце семидесятых прошлого века, стало очевидным и свойство, которое так сильно — и технологически, и экономически — отличает программные продукты от продуктов других производств. Это свойство **безущербного копирования**.

Что нужно сделать, чтобы вместо единственной хорошей детали стало две одинаковых? Купить чугунную болванку, а рабочий выточит из неё деталь. Вторая деталь нам ни к чему, мы её отдали; чего мы при этом лишаемся, ведь первая-то при нас? Разумеется, денег: болванку мы покупали, а токарю — платили, это его труд создал копию. Копирование причиняет ущерб, который, по-честному, надо возмещать. Например, тем, что детали отныне мы будем не дарить, а продавать.

Так. А что нужно для того, чтобы из единственной хорошей программы стало две одинаковых? Выполнить операцию копирования, больше ничего. Если тот, кому вы собираетесь эту программу подарить, принесёт с собой записываемый лазерный диск, вам это ни во что не станет. Копирование экземпляра программы **безущербное** для хозяина этого экземпляра.

Тут между разработчиками программ случился раскол. Одни (будем называть их «эгоистами») говорили: «надо запретить свободное

распространение не только исходных текстов, но и вообще самих программ. Программы надо продавать! Это ж сколько денег можно выудить прямо из воздуха. А вы, господа альтруисты (назовём их так), просто олухи царя небесного, маргиналы, пиджаков не носите, питаетесь кофе и гамбургерами, и ничего не знаете о том, как вести бизнес».

«Альтруисты» возражали: «нам всё равно, что носить, мы любим кофе, и это наши программы, мы их написали, мы хозяева — что хотим, то и делаем. А если мы не будем свободно распространять исходные тексты, как вообще тогда программы разрабатывать? В одиночку, что ли?»

«Зачем в одиночку? — напирали эгоисты, — из воздуха-то можно много денег понаделать, мы наймём вам помощников, сколько надо, только не показывайте никому трудов своих, а для пущей крепости хозяевами ваших программ будем считаться мы, а уж мы-то позаботимся о том, чтобы каждая копия превращалась в зелёную бумажку».

Мировое сообщество в прошлом веке в целом знать не знало ни о каком безущербном копировании, так что с удовольствием начало вырабатывать законы, это копирование запрещающие. Главный ход, конечно, состоит в том, что право распоряжаться такой вот «эгоистской» программой (в отличие от чугунной детали) должно всегда оставаться у одного человека — правовладельца. Он и будет получать доход с продажи, его права и защищают законы.

Это нужно знать, между прочим. Допустим, вы пошли в магазин, и купили там втридорога лазерный диск с копией «операционной системы» или каким-нибудь другим программным продуктом, предоставляемым без исходных текстов и без возможности их получить. Ущерб, который потерпит ваш карман от такого безущербного копирования, наводит на мысль о том, что ваша копия сопровождается правовладельческой лицензией. По сути дела, обладая копией — экземпляром — программы, вы *не становитесь хозяином* программы как произведения. Вам бы дальнейшее её распространение не причинило никакого ущерба, но права на это у вас нет, такое право сохраняется только за «продавцом» или другим правовладельцем программы.

Правовладельческие программы называют также **ПО ЗК** — Программное Обеспечение с Закрытым Кодом, потому что исходный текст этих программ использовать запрещено, а чаще всего и смотреть на него нельзя, или «проприетарными» (от proprietary, «собственнический»), что отражает правовладельческую суть дела.

Свобода программного обеспечения и GNU Public License

Люди, названные нами «альтруистами», — программисты университетской школы и вообще люди, ищущие интереса прежде выгоды, поначалу просто не обращали внимания на воцаряющийся правовладельческий строй. Они продолжали свободно обмениваться текстами и совместно дорабатывать программы друг друга. Однако чувствовали они себя неуютно, и неспроста. В начале восьмидесятых последовала серия громких судебных процессов по алгоритму «ты написал принадлежащую мне программу и украл её у меня, передав товарищу». Беспокойство нарастало.

Между тем стало вполне понятно, что именно отличает традиционный свободный способ существования программы от «правовладельческого». Достаточно, чтобы любой, кто получил копию программы, вместе с ней получал *те же* права, что и сам автор, в первую очередь право её изменять и копировать. При этом *каждый* становится «хозяином программы как произведения». Тогда можно будет с чистой совестью продолжать работать совместно и свободно: тиражировать программу, продавать её, раздавать за так, улучшать и ухудшать, тиражировать то, что получилось, и т. д.

Принципы свободы программного обеспечения несколько позже сформулировал видный деятель Фонда Свободных Программ (FSF) Ричард Столлман:

- Свобода запускать программу в любых целях
- Свобода изучать работу программы и вносить в неё изменения
- Свобода распространять копии программы
- Свобода публиковать внесённые изменения и распространять копии изменённой программы

Смысл этих принципов ясен: во-первых, пользоваться, во-вторых исправлять то, что не нравится, в-третьих, привлекать всех, кто пожелает, в-четвёртых, работать совместно. Стоит один из четырёх принципов нарушить, и работать над программой сообща будет невозможно.

Перечисленные четыре принципа свободы ничем не ограничивают потенциального хозяина копии программного продукта. Он может делать с этой копией что угодно, в том числе *нарушать* своими действиями хоть все четыре принципа. Скажем, исправлять ошибки и распространять результат с правовладельческой лицензией, то есть

объявить себя хозяином всех последующих копий. Исправления и исходный текст исправленной программы он, конечно, никому не откроет — это «ноу-хау», рычаг для добычи прибыли из воздуха. Немало в прошлом свободных программ и их частей заключено нынче во мрак несвободных программных продуктов. Плохо здесь совсем не то, что кто-то зарабатывает на чужом, в сущности, труде; зарабатывает — это хорошо!

Плохо, что общественной пользы от такого сокрытия никакой: никто не узнает, что это были за ошибки; свой талант человек, считай, в землю зарыл — в надежде, что вырастет золотое дерево. Эффективность разработки при этом — мизерная: второму такому же умнику придётся начинать с *того же самого* места, с открытого исходного кода, и третьему, и десятому. И вот сидят они по своим углам, никому ничего не говорят, и исправляют одни и те же ошибки без надежды поделиться результатами успеха с коллегами.

После нескольких болезненных судебных столкновений с правовладельческими лицензиями по схеме «программировал-то ты, а хозяин — я», Ричард Столлман и его коллеги из сообщества профессиональных программистов GNU разработали собственную *лицензию* — соглашение между хозяином копии программы и предполагаемым получателем этой копии. Она получила название GNU Public License (Общественная Лицензия GNU). Только после принятия условий этой лицензии человек может считаться хозяином полученной им копии программного продукта и исходных текстов.

Общественная Лицензия GNU декларирует пресловутые «четыре свободы», которые гарантированы хозяину копии, содержит некоторые юридические тонкости относительно их соблюдения и — главное — накладывает на хозяина единственное строгое ограничение: если новоиспечённый хозяин захочет распространять программу или изменённый её вариант, условия её распространения должны быть *не хуже* GPL, то есть включать в себя «четыре свободы» и вот это самое требование их дальнейшего соблюдения. Программисту, знакомому с понятием «наследование» идея вполне прозрачна, законникам пришлось объяснять, а со стороны это выглядит совсем просто: требования свободы нельзя нарушать, и всё. Чаще всего проще не мучиться, изобретая собственные лицензии, а публиковать плоды совместных трудов по той же GPL.

GPL не имеет прямого отношения к сути свободного ПО, это *инструмент*, оружие, которое в мире правовладельческих лицензий помогает сообществу отстаивать право на свободу.

Сообщество кого?

Важнейшее прикладное следствие свободы программного продукта — свободная совместная его разработка. Всякий, кто хочет и может поучаствовать в общем деле, может приступать к работе без каких-либо не имеющих отношения к вопросу ограничений. В каком качестве заинтересованный человек может присоединиться к сообществу?

Ядро (Core Team)

Во-первых, ответственный и квалифицированный человек может взять на себя часть разработки программного продукта, и, что гораздо важнее и сложнее, принятие решений по всем принципиальным вопросам. Таких людей, как правило, совсем немного, а доля их участия в жизни продукта велика. Неважно, почему они согласны тянуть лямку впереди всех, важно, что они это делают, имеют достаточно рабочего времени и личных достоинств. Они составляют **ядро** сообщества. Как правило, это умудрённые опытом программисты или эксперты, словом, те, к чьему мнению заслуженно прислушиваются. Именно от мнения ядра зависит, как будет развиваться продукт, как и когда вносить в него предложенные поправки, когда и как делать выпуски (release) и т. п.

Ядро соответствует понятию «команда разработчиков» старой, индивидуалистической схемы. Здесь более или менее срабатывают и практические рекомендации старой школы: ядро должно быть небольшим, чтобы могло само с собой договариваться, должно иметь разделение труда, чтобы каждый занимался в первую очередь тем, к чему имеет больший талант, должно иметь «главного» для волевого разрешения неразрешимых вопросов и т. п., словом, типичный «team» из пяти-семи человек.

Сообщество разработчиков (Development Team)

Сообщество разработчиков — самая главная часть любого свободного программного продукта. Дело в том, что полдюжины человек в ядре, конечно, не могут *слишком* быстро заниматься разработкой. Что ещё важнее — они не могут в одиночку управляться со всей эксплуатационной историей, то есть отслеживать, как, когда и с каким успехом применялась программа, какие были ошибки, были ли они исправлены

и как. Именно свобода ПО предоставляет любому, кто поучаствовал в эксплуатационной истории, сделать эту историю публичной и внести её в русло общей работы.

Самый простой способ подключиться к сообществу — найти ошибку в программе, исправить её и написать разработчикам. Вполне вероятно, что вы, в силу специфики работы, наткнулись на эту ошибку впервые. Программа свободная, исходные тексты имеются, право на публичное её изменение — тоже, так что толковый программист в состоянии ошибку найти и исправить. Если он при этом будет настолько ленив и неблагодарен, что не сообщит авторам, всё сообщество укажет на него с осуждением. Если узнает, конечно.

Что всего приятнее — практически любой человек, мало-мальски сведущий в программировании, может поправить ошибку. Титаном мысли для этого можно и не быть. Если предложенная тобою заплата (patch) написана из рук вон плохо, ответственный разработчик из Core Team предпочтёт скорее переписать это место наново, чем выбросить и забыть, на то он и ответственный.

Свободные проекты очень много внимания и ресурсов уделяют поддержке сообщества разработчиков. Техническая документация по проекту (руководство по сборке и установке), как правило, не уступает пользовательской. Для разработчиков заводятся специальные списки почтовой рассылки, в которых они обмениваются информацией друг с другом и задают вопросы, и — что тоже очень важно — автоматизированные системы отслеживания ошибок и предложений (т. н. «bug tracker»-системы).

Эта политика весьма эффективна ещё и вот в каком плане. Программист, вместо того, чтобы кидаться к клавиатуре и начинать программировать порученный ему продукт с нуля, берётся за мышь и педантично обшаривает такие ресурсы, как SourceForge, FreshMeat или NonGNU, на предмет подходящего свободного проекта, который решает те же задачи. С очень высокой степенью вероятности такой проект найдётся, хотя, возможно, и не будет обладать какими-то специфическими для конкретной ситуации способностями. Всё что останется программисту — это модифицировать свободный продукт, дополнив его этими возможностями. Добросовестный и благодарный человек не преминет поделиться с сообществом, а от происков любителей правовладельческого лицензирования его защищает GPL: если программу вообще собираются распространять, исходный код необходимо открыть, а уж от этого до интеграции изменений ядром разработчиков в основной проект дорожка прямая.

Само собой, этот Добросовестный Программист автоматически становится членом сообщества разработчиков, что, помимо прочего, сулит и некоторые служебные перспективы.

Сообщество пользователей (Users Community)

Бессмысленно говорить о каком-то программном продукте, не упоминая тех, кто им будет пользоваться. Старая «хакерская» практика — разработчик и пользователь суть одно лицо, потому что программировать гораздо интереснее, чем запускать программу — преобразовалась за эти годы до неузнаваемости. Мы не знаем, почему те или иные члены сообщества разработчиков занимаются нашей программой — из-за денег, славы, спортивного интереса, производственной необходимости или на голом энтузиазме; они работают — и огромное им спасибо. Но почему мы, пользователи, хотим видеть программу мощной, толковой и надёжной — совершенно понятно: программа нам *нужна*.

Конечно, пользователь-программист или программист-пользователь продолжают играть немаловажную роль в развитии свободного ПО. Такой человек в сообществе решает сразу три задачи: активно тестирует продукт, находя ошибки и выдумывая, что ещё «в супе не хватает», сам формулирует программистское задание, и сам же его выполняет. Поэтому сообществу всегда выгодно привлечь активного пользователя в ряды разработчиков либо убедить талантливого программиста пользоваться свободным ПО.

Но уверенно надеяться на чудесное слияние столь разных ролей можно только в редких случаях «программирования для программирования», чаще всего пользователем может быть кто угодно, а иногда профессия типичного пользователя *не* позволяет ему быть программистом. Например, пользователь компилятора — скорее всего программист, музыкального проигрывателя — совсем неважно кто, а пользователь кассового аппарата скорее всего имеет самое смутное представление о том, каким образом аппарат цифры показывает.

В конце концов, трёхступенчатый процесс поиска эксплуатационных трудностей, формулировки задачи и программирования давно уже освоен «классической» схемой разработки: пользователь отыскивает повод для улучшения, пользователь и программист совместно формулируют задачу, программист её решает. И от программиста, и от пользователя в этом случае требуется желание, возможность и умение взаимодействовать. Не очень-то и редкие качества, если учитывать, что человек — животное общественное и вообще склонен общаться с

себе подобными. Однако даже это может стать камнем преткновения: несвободный, «закрытый» способ разработки пошёл по экстенсивному пути развития; для решения каждой задачи нанимается специальный отдельный человек (тестер, консультант, менеджер проекта).

Преткновение как раз в том, в чём закрытый способ разработки отказывает решительно всем, даже самим пользователям: в *заинтересованности*. Общественная схема начинает работать только тогда, когда и пользователь, и разработчик хотят (неважно, по каким причинам) *решать* возникшую задачу (а не только «чтобы была решена»). Но ведь сообщество образуется как раз из заинтересованных, из тех, кому для работы не жалко трудов. Дополнительное требование — пользователь и разработчик должны быть *в состоянии* решить задачу, то есть иметь достаточную квалификацию каждый в своей области. Сообщество вокруг свободного ПО — не для неучей, что правда, то правда. Впрочем, мало найдётся людей, столь невосприимчивых к знаниям, чтобы остаться неучами вопреки активному интересу и всеобщему образованию.

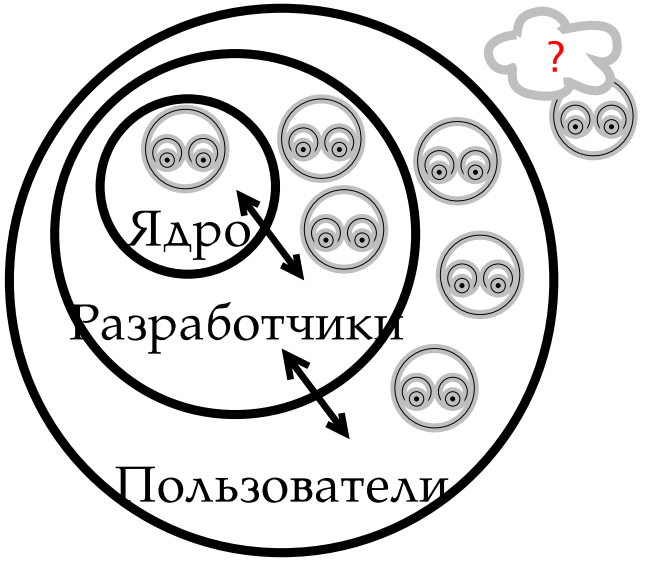

Рис. 5.1. Сообщество вокруг свободного программного продукта

Единое информационное пространство

Для того, чтобы приведённая выше схема заработала, одного желания мало. Недостаточно и умения. Делать что-то *сообща* можно, только если есть *возможность* сообщать и общаться. Передавать друг другу информацию — во-первых, и ориентироваться в ней — во-вторых.

В самом деле. Группы в сообществе — ядро, разработчики и пользователи — представляются эдакими тремя комнатами, где каждый знаком с каждым, все затруднения активно обсуждаются между собой, а двери из комнаты в комнату никогда не закрываются: всякий результат обсуждения нужно довести до сведения товарищей по сообществу, да и просто так поговорить, чтобы быть в курсе, не мешает. А ведь в действительности даже ядро сообщества может состоять из людей, которые живут на различных континентах, и лично каждый не с каждым и знаком. Что и говорить о всемирном сообществе разработчиков или пользователей?

Переоценить вклад глобальных сетей в развитие свободного ПО невозможно. Именно за счёт свободного распространения информации людям не обязательно постоянно находиться в одной комнате в обществе друг друга. Именно за счёт связности сети Интернет все пользователи и разработчики имеют *возможность* собираться вместе, обсуждать проблемы, обмениваться новостями, узнавать друг о друге, словом, вести себя так, как если бы все находились в «одной комнате».

Ещё важнее простота, оперативность и надёжность связи *между* группами, без которой не может жить никакой проект свободного ПО. Если на схеме убрать стрелочки, она превратится в три беспомощных замкнутых множества, в каждом из которых будет куча нерешаемых задач. Ядро не сможет быстро и оперативно разрабатывать программу, внешние разработчики — договориться о том, какие именно задачи решать, куда идти, да и решать ли вообще, а пользователи — добиться удобной и безошибочной работы программы.

Живые люди

Прежде, чем говорить о *технических* возможностях, которые обеспечивают взаимодействие людей в сообществе, стоит сказать пару слов о том, зачем это техническое обеспечение им нужно.

Необходимость общения в сообществе очевидна (сами слова-то почти одинаковые). Столь же очевидно, что никто не сможет заставить члена сообщества общаться, кроме него самого, кроме желания побольше узнать и помочь. С другой стороны, редко у кого такое желание — часть *профессии*. Иными словами, за жажду знаний, как и за миссионерство, нечасто платят. Это означает, что сам *способ* общения и приёма-передачи знаний должен быть настолько прост и вместе с тем соблазнителен, чтобы им добровольно стало пользоваться как можно больше хороших и нужных людей.

Живых людей, которыми движет *желание* работать и *необходимость* общаться (а не наоборот). Из чего следует, что чем удобнее *инструмент* и *структура* обмена информацией, тем проще уступить желанию, смирившись с необходимостью. (В скобках заметим, что закрытый путь разработки решает эту проблему привычным экстенсивным способом: нанимаются специалисты по PR, профессиональные проповедники и автоответчики во плоти, обязанность которых — отвечать на вопросы, даже если в ответах нет никакого смысла).

Кузница, Горнило и Багзилла

Каналы связи *между* группами существуют и сами по себе — за счёт пользователей-разработчиков, которых в свободных проектах немало. Однако этого крайне недостаточно. Как всякий труд, совместная разработка требует дисциплины и формализации, причём чем больше в её составе людей и чем меньше между ними социальных связей, тем строже должна быть дисциплина и требовательнее формальность. В случае свободного сообщества, когда коллеги по проекту могут не быть даже знакомы, вопрос дисциплины и формы встаёт особенно остро: без этого культурные, социальные, психологические и прочие различия возьмут верх, и результатом будет неуправляемое вече. С другой стороны, нельзя перегибать палку, иначе свободно пришедший в сообщество человек плюнет на все формальности и свободно уйдёт.

Структура

Первым делом сообществу стоит договориться о том, кто какие права и возможности имеет, и каким образом их осуществляет. Где хранятся исходные тексты программы и согласно какой дисциплине их менять. Как принимать сообщения об ошибках и реагировать на них. Какая документация существует по программному продукту, где её взять и как с ней работать. Что необходимо знать, прежде чем включаться в сообщество. Где, наконец, находятся *готовые* версии программного продукта, и чем одна отличается от другой.

Вопросов много, и способы ответа на каждый из них мировое сообщество нащупывало десятилетиями. Дело осложнялось тем, что разработчики — движущая сила любого сообщества — редко когда оказывались толковыми веб-дизайнерами. Не в смысле умения проектировать что-либо, а в смысле умения это наглядно показывать. Впрочем, со временем необходимость выдумывать что-то совсем уж своё отпала, потому что выработались эффективные шаблоны организации труда. Тогда-то и стали появляться сетевые проекты, которые предлагают уже готовый набор таких шаблонов — **порталы**. Сообществу остаётся только согласиться использовать этот набор в работе и наполнить шаблоны содержательной информацией.

Так, например, устроен упомянутый выше проект Source Forge: система хранения и обновления исходных текстов, хранилище готовых программ, система отслеживания ошибок, форумы для обсуждения, списки рассылки и т. д. — всё для успешной организации сообщества. Зарегистрироваться, положить первые десятки строк программы — и пошло-поехало. Кстати, на сегодня в SF зарегистрировано больше десяти тысяч проектов и много больше миллиона пользователей.

Лицо

Что меньше всего стоит передоверять универсальным службам, вроде SF, так это — эстетическую и идеологическую составляющую любого сообщества. В конце концов, технические средства — это всего только инструмент, а как им распорядиться решает, ядро при поддержке всех единомышленников. Немалую роль здесь играет всем уже сегодня привычное понятие «сайт» — сетевой WWW-ресурс, посвящённый самому программному продукту, его задачам, способам использования, дисциплине разработки и т. п. От того, как устроен сайт проекта, в немалой степени зависит и интерес новых посетителей сайта, и активность самого сообщества. Сайт — это лицо проекта.

Сайт совсем не обязан быть огромным и сложным. Скорее наоборот: чем легче будет посетителю ориентироваться, тем выше вероятность, что он найдёт себе подходящее место в сообществе. На сайте обычно лежат материалы трёх видов. Во-первых, научно-популярные и идеологические, которые описывают, зачем проект нужен самим его

создателям, и чем он может помочь пользователям. Во-вторых — информационная часть с пользовательской и технической документацией, что делает включение в сообщество вопросом чисто техническим. И в-третьих — технически-структурная часть, которая описывает все элементы взаимодействия в сообществе, а также отсылает к используемым ресурсам. Нередки ситуации, когда несколько титульных страниц проекта, особенно небольшого, — дело рук разработчиков, а всю структурную и ресурсную часть (хранение исходных текстов и готовых программ, отслеживание ошибок, информационные рассылки и прочее) берёт на себя какой-нибудь портал.

Обратная связь

Несколько раз уже упомянутая служба отслеживания ошибок — чуть ли не самый важный из механизмов, формирующих сообщество. Дело в том, что распространение свободных программ, равно как и информации о них, через сайт — дело благое, но чрезвычайно неблагодарное, если не предусмотреть *обратной связи* от тех, кто эту информацию усвоил, а программу использует. Стрелки на схеме *обязаны* быть двунаправленными, иначе ядро не получит никакой выгоды от сообщества разработчиков, и все они вместе — от сообщества пользователей.

Чтобы сообщить об ошибке разработчику, необходимо иметь: ошибку, разработчика, и — главное! — *механизм*, позволяющий оформить подобное сообщение. Причём в случае распределённой совместной работы этот механизм должен сопровождаться известной дисциплиной оформления, чтобы первый попавшийся разгильдяй не принялся докучать ответственному разработчику посланиями вроде «Дорогие учёные. У меня который год в подполе происходит подземный стук. Объясните, пожалуйста, как он происходит». В неменьшей степени это относится и к прямой работе сообщества над исходными текстами программы.

Стоит ещё раз подчеркнуть, что средства совместной разработки должны быть в первую очередь удобны технически, то есть самим разработчикам и активным пользователям. Именно с этим связано разнообразие и солидная история как систем отслеживания ошибок, так и систем совместного написания текста программ. В числе первых назовём GNATS, BTS, BugZilla, Mantis, Eventum и множество других, как родившихся в недрах больших сообществ для собственных нужд, так и специально предназначенных для шаблонов совместной

работы любого сообщества. Системы совместного ведения архива исходных текстов традиционно называются системами «контроля версий» (version control), хотя их возможности давно уже этим не исчерпываются. Среди них — всемирно популярная CVS, её наследники SubVersion и GNU Arch, специализированные на особые задачи git или monotone, крайне простые (darcs) и весьма мощные (вплоть до поддержки автоматической сборки получившейся программы — aegis), и некоторые другие.

Всякому, кто хочет включиться в сообщество разработчиков, придётся научиться взаимодействовать и с тем, и с другим. Только так можно сохранить относительный порядок при полной свободе добровольно собравшихся вместе людей.

Списки, Блоги и Вики

Для организации взаимодействия *внутри* группы, особенно вне рамок исходных текстов, нужны другие средства. Нужно то самое произвольное общение, которое привиделось в форме «трёх комнат с открытыми дверями». Где формальная сторона была бы сведена к минимуму, а преобладала бы, насколько это возможно, общечеловеческая.

Лет пятнадцать-двадцать назад было трудно себе представить, что в обмене символами посредством компьютера есть так уж много человеческого. Бытовало мнение, что электронная почта — удел профессионалов-компьютерщиков, уже не различающих электронную и явную реальности, а, скажем, сетевой дневник (web log, или «the blog») — это такая форма электронного эксгибиционизма полностью исключённого из «настоящего» социума киберпанка. Скорей уж инструментом, приближающим электронное к человеческому, мыслился ужасный электрод, вставляемый непосредственно в голову и навевающий электронные сны, либо передающий мысли по проводам. Что было и продолжает оставаться фантастикой. Гора не шла к Магомету.

Сейчас мы стали свидетелями того, как охотно Магомет, то есть современное человечество, двинулся к горе. Электронная почта заменила бумажную. Слово «форум» вызывает стойкие ассоциации с веб-сайтом, и только после некоторого напряжения вспоминаешь его словарное значение. В последнее время наблюдается взрыв увлечения электронными дневниками (не в последнюю очередь, кстати, из-за того, что исходный текст «движков» некоторых из них открыт, и теперь каждый может основать свой сайт с дневниками). Купить вещь

в Интернет-магазине с доставкой на дом зачастую дешевле, чем влачиться за нею куда-то на склад, не говоря уже о посещении чистого и сверкающего, но людного и дорогого магазина.

Первое, что было придумано, а скорее — возникло само собой для информационного сплочения сообщества — тематические списки рассылки[1]. Как правило, каждое сообщество имеет несколько таких списков — для разработчиков, для пользователей, для обсуждения стратегических вопросов, технические, дизайнерские, наконец, для «роботов», собирающих статистику; крупные проекты, например, дистрибутивы, имеют их до сотни. Список рассылки — то самое место, куда надо задавать вопросы: если человек читает рассылку, ему это для чего-нибудь нужно. Он поможет вам, а когда само попадёт впросак, кто-нибудь поможет ему.

Для общения в совсем необязательном режиме — трёпа, тусовки и прочего, что в цифровом мире заменяет совместное времяпрепровождение — используются пресловутые «форумы», сетевые журналы и обмен текстами в реальном времени (chat). Это ресурсы уже не совсем сообщества, а, так сказать, пригород его, тропки, которыми можно в сообщество прийти. А можно и не прийти, если процесс обмена словами важнее совместной работы. Так что эти форумы — ещё и фильтр ответственного отношения к делу. Гулякам праздным там живётся лучше, чем в дисциплинированной среде сообщества.

Первое исключение из этого правила: ресурсы Internet Relay Chat (irc), которые достаточно стары, как следствие — слегка сложноваты для освоения гуляками и имеют свои культурные традиции защиты от дурака. IRC-каналы часто используют профессионалы для оперативного решения задач в стиле «вопрос-ответ». Второе: всё чаще дневники (Blogger, Live Journal и т. п.) стали использоваться с самыми серьёзными целями: информация для сообщества, обсуждение принципиальных вопросов и т. п. Причина этого — особый способ доступа к информации в сетевых дневниках, когда пользователь читает не то, что ему захотели прислать, а те источники информации, что он предварительно согласился читать. Получается отличный фильтр действительно заинтересованных, что, собственно, и надо сообществу.

Наконец, в последнее время набирает обороты практика совместной разработки сайтов. Эта практика и сопутствующая ей технология получила название «Wiki» (от гавайского wiki-wiki, «быстро-быстро»).

[1]Наверное, уже и не надо объяснять, что такое «список рассылки»? Это такая форма электронной почты, когда один человек её отправляет, а несколько — подписчики — получают.

Практика в том, чтобы сделать процесс наполнения сайта содержимым *очень* простым. Настолько простым, чтобы туда написать мог *кто угодно*, было бы желание. Ответственному за сайт придётся только просматривать новые поступления на предмет очевидного хулиганства (пресечь которое в Wiki-технологии достаточно просто) и, возможно, самому добавлять страницы структурирующего характера (чтобы сайт не превращался в свалку бессвязных текстов). Несмотря на молодость технологии, большинство крупных проектов уже обзавелось своими Wiki — официальными и неофициальными, которые служат отличным хранилищем данных и поисковой системой по ним. Надо ли говорить, что качество и наполнение Wiki-сайта целиком зависит от усилий сообщества?

Что такое хорошо и что такое плохо

Условия успеха

Сообщество вокруг программного продукта добивается успехов, если созданы условия для совместной работы: свободное распространение, три части сообщества — ответственная (ядро), компетентная (разработчики) и заинтересованная (пользователи), доступное и толковое информационное пространство, в первую очередь — простые и надёжные каналы связи в группах и между ними.

Тогда каждый занимает в сообществе место по силам и желанию, и делает только то, что может и должен. Чем грамотнее спланировано информационное пространство, тем меньше накладных расходов и тем больше приходит компетентных людей, которые по тем или иным причинам готовы развивать программный продукт. Проект, фактически, пишет сам себя, остаётся только направлять его движение и следить за тем, чтобы сообществу хорошо жилось.

Конечно, в настоящей жизни бывает всякое, любая правовая, политическая или экономическая неприятность может пагубно сказаться на жизни программы, но успех крупных свободных программных проектов в наибольшей степени зависит от умения управлять сообществом.

За чертой остаются все, кто по какой-то причине не может включиться в сообщество. Это не означает, что, изолируясь от сообщества, человек не может получить определённую выгоду от использования свободного программного продукта. Большинство свободных программ вполне «дистрибутивны»: их распространяют и используют

как по отдельности, так и в составе **дистрибутива** — тематического или системного сборника программ, с помощью которых пользователь может решать задачи определённого направления. Как правило, дистрибутив даже единственного программного продукта содержит достаточно информации, чтобы пользоваться им автономно. Но в этом случае пользователь лишается *главного* преимущества — поддержки сообщества, непосредственной помощи со стороны знающих людей, которую невозможно ни запрограммировать, ни задокументировать.

К тому же любая толковая просьба о помощи — сама по себе уже *помощь* сообществу! Если вопрос возник, а ответа на него нет ни в документации, ни в списках рассылки, скоро с такими же трудностями столкнутся и другие пользователи. Так отчего бы не прояснить вопрос до того, как станет неудобно многим? А сообщение об ошибке или короткая методичка по найденному вами решению задачи — это уже *услуга* сообществу, которую легко и приятно оказать в корыстных целях: ошибка будет исправлена, а ваш текст опубликован. Взаимодействие с сообществом — это *естественный* способ существования пользователя свободного ПО; если, например, вы страстно хотите каких-либо изменений в программе, но боитесь в этом признаться, вы их не дождётесь. А если вы их изложите сообществу, да подкрепите просьбу добрым словом, пивом, сотней рублей, сотней долларов — чем-нибудь вполне адекватным объёму работ, ответ не заставит себя ждать.

Граждане и гражданки! Проявляйте социальную активность! Включайтесь в сообщество пользователей!

Унесённые призраками

Теперь нетрудно понять, когда разработка ПО по открытой схеме себя не оправдывает. Как только какое-нибудь из перечисленных «условий успеха» нарушается, ограничивается одна из свобод, сразу возникают затруднения, подчас — совершенно непреодолимые в рамках свободной модели.

Нашлись какие-то правовладельческие документы и соблюдать «четыре свободы Столлмана» нельзя? Прощай, открытая совместная разработка! Только своими силами, да ещё и подписку о неразглашении давать придётся.

В стране запрещён или не поощряется Интернет и вообще открытая передача данных? Тогда нет смысла в совместном проекте — без

обратной-то связи. Не случайно в Китае так много сильных программистов и хакеров и так мало сообществ свободного ПО (да есть ли они вообще?).

Если пользователи не заинтересованы в самой программе, работают из-под палки или просто ничего не понимают и не хотят понимать, возникает огромный зазор между ними и разработчиками. Тогда каждый, кто и пользователь и разработчик одновременно, бесценен. А таких может не оказаться. Например, до недавнего времени инструментарий *менеджера* — программные средства управления персоналом, сетевого менеджмента и т. п. — был представлен в свободном ПО очень скудно. По причине, как это ни смешно, застарелой нелюбви разработчиков и управленцев друг к другу. Только в последние лет пять свободное ПО начало входить и на этот рынок.

Другой подобный пример — компьютерные игры, где пропасть, разделяющая разработчика и пользователя-игрока преодолевается только на могучей волне энтузиазма; а чаще — не преодолевается и замыкается в ПО ЗК. Правда, и здесь наметились сдвиги: архитектуру компьютерной игры («движок») теперь иногда делают открытой, надеясь на помощь сообщества в главном — в наполнении.

Если потенциальных пользователей очень мало, скажем, дюжина человек на весь мир, то открытость или закрытость разработки никакой роли не играет: всё равно придётся работать с каждым индивидуально, а ждать чудесного самозарождения сообщества разработчиков вообще не приходится. С другой стороны, здесь могут сыграть как раз индивидуальные пристрастия: в научном мире, к которому вполне могут относиться эти двенадцать пользователей, принято открыто обмениваться если не всей информацией, то уж точно — инструментами (а программа для учёного — именно инструмент).

Если в проекте большая «расходная статья» на непрограммистские задачи (скажем, много работы для художника, электронщика, сценариста и т. п.), а «доходная статья» (техническое обслуживание, заказная доработка, обучение и т. п.) — маленькая или вообще не предусмотрена, делать его свободным, к сожалению, невыгодно. Ситуация, характерная опять-таки для больших компьютерных игр — и для узкопрофильных разработок, где оплачивается участие многих специалистов, а возможности пользовательского сообщества невелики.

И всё-таки главное препятствие развитию свободного ПО — у нас в головах. Человечество тысячелетиями вырабатывало культуру права собственности, имея перед собой исключительно материальные объекты, копирование которых требует затрат. Объекты нематериальные,

«технологии», появились в массовом порядке менее двух веков назад, а «программным продуктам», безущербность копирования которых очевидна, ещё не скоро исполнится полвека. Неудивительно, что люди, которые *сами не программируют* (управленцы, экономисты, юристы и прочие), всё ещё не готовы отличить собственность на чугунную болванку от собственности на программу. Культура нематериальной собственности только вырабатывается.

Напоследок

Время одиночек, которые самостоятельно, от начала до конца, прорабатывают весь программный продукт, прошло. Для того, чтобы написать хорошую программу, требуются усилия *различных* специалистов: от системного архитектора до кодировщика и от художника до эксперта в прикладной области. Собрать всех этих людей воедино можно либо на добровольной, свободной основе, либо «кнутом и пряником» — деньгами и крепостной зависимостью, правовладением.

Правовладельческая «команда» устроена так: неважно, как человек изначально относится к своим обязанностям и что думает, главное — платить ему и создавать условия для эффективной работы. Тогда он либо начнёт приносить пользу, либо лишится места. Ущерб, который он в этом случае может нанести компании, надо ограничить запретительной лицензией.

Свободное сообщество устроено строго наоборот: неважно, *почему* человек хочет участвовать в разработке, важно, чтобы он *хотел* этого, мог быть полезен и относился к работе ответственно. Ущерб, который может быть нанесён сообществу путём сокрытия информации, ограничивается открытой лицензией GPL.

Важно помнить, что свобода — это в первую очередь ответственность. И дело не только в том, что в свободном мире каждый обязан отвечать за свою работу и не может свалить ответственность на другого. В свободном мире безответственные действия не приносят выгоды. Например, крахом обернётся попытка продавать воздух в стиле ПО ЗК: любой свободный продукт стоит столько, сколько заслуживает, как это и полагается настоящими требованиями рынка; а если искусственно завышать цену одного экземпляра, всегда найдутся желающие приобрести и продавать его по цене естественной.

Что же приносит доход в мире свободного ПО? А что везде: труд, оперативность, информация, качество.

`Доработка`

Свободное ПО хорошо тем, что *кто угодно* может адаптировать его под свои нужды. Или заплатить кому-нибудь из сообщества разработчиков за такую адаптацию. Причём чем качественней продукт, чем больше вокруг него сообщество – тем *меньше* для самого сообщества себестоимость доработки, тем *больше* разница между суммой, которую готов заплатить клиент, и затратами.

`Заказная разработка`

Сообщество свободного ПО — своего рода биржа труда профессиональных разработчиков. Она имеет свою специфику (например, попадают туда не от безработицы, а напротив, вследствие работы, отчего народ там разборчив к предложениям), грамотно манипулируя которой можно в короткий срок получить в своё распоряжение работоспособную команду с приличной профессиональной историей. Главное — не нарушать принципов их свободы.

`Техническая поддержка`

Услуги — работа компетентного человека здесь и сейчас — во всём мире и во все времена ценились высоко. Кому же знать специфику программного продукта, как не активному члену сообщества? Частенько человек *работает* в команде разработчиков близко к ядру, а то и вовсе в нём, а *зарабатывает* на жизнь, скажем, системным администрированием.

`Консультация и обучение`

То же самое относится и к услугам, так сказать, социального плана. В каждом сообществе наверняка находится человек, не только сведущий, но и умеющий грамотно изложить свои сведения — устно ли, письменно ли. Таких людей немного и оплачивается их труд довольно высоко. О том, какую пользу они приносят сообществу, строя то самое вожделенное информационное пространство, и говорить не надо.

Время одиночек, наверное, прошло. Прошло и время вождей, ведущих за собою толпы, неведомо куда под неведомо какими лозунгами. Человек — свободная личность — хочет выбирать свой путь самостоятельно и следовать своему собственному пути, опираясь на опыт и помощь других свободных людей. Не такая уж и малая, в сущности, часть общества — пользователи и разработчики свободных про-

грамм — подают пример такого свободного взаимодействия. Возможно, этот пример нельзя скопировать на произвольную повседневность. Даже наверняка нельзя. Повседневность каждого из нас — если, конечно, это наш свободный выбор — невозможно скопировать. Это как программа, написанная лично тобой.

Тем не менее этот пример актуален. Свободные программы для свободных людей.

Глава 6

Больше, чем дистрибутив

Система управления пакетами APT

Александр Боковой, Дмитрий Левин, Кирилл Маслинский

Введение: пакеты, зависимости и репозитории

В современных системах на базе Linux огромное число общих ресурсов, которыми пользуются сразу несколько программ: разделяемых библиотек, содержащих стандартные функции, исполняемых файлов, сценариев и стандартных утилит и т. д. Удаление или изменение версии одного из составляющих систему компонентов может повлечь неработоспособность других, связанных с ним компонентов, или даже вывести из строя всю систему. В контексте системного администрирования проблемы такого рода называют нарушением **целостности системы**. Задача администратора — обеспечить наличие в системе согласованных версий всех необходимых программных компонентов (обеспечение целостности системы).

Для установки, удаления и обновления программ и поддержания целостности системы в Linux в первую очередь стали использоваться **менеджеры пакетов** (такие, как RPM в дистрибутивах RedHat или dpkg в Debian GNU/Linux). С точки зрения менеджера пакетов программное обеспечение представляет собой набор компонентов — программных **пакетов**. Такие компоненты содержат в себе набор исполняемых программ и вспомогательных файлов, необходимых для корректной работы ПО. Менеджеры пакетов облегчают установку программ: они позволяют проверить наличие необходимых для работы устанавливаемой программы компонент подходящей версии непосредственно в момент установки, а также производят необходимые процедуры для

регистрации программы во всех операционных средах пользователя. Сразу после установки программа оказывается доступна пользователю из командной строки и появляется в меню всех графических оболочек.

Благодаря менеджерам пакетов, пользователю Linux обычно не требуется непосредственно обращаться к установочным процедурам отдельных программ или непосредственно работать с каталогами, в которых установлены исполняемые файлы и компоненты программ (обычно это `/usr/bin`, `/usr/share/имя_пакета`) — всю работу делает менеджер пакетов. Поэтому установку, обновление и удаление программ в Linux обычно называют управлением пакетами.

Часто компоненты, используемые различными программами, выделяют в отдельные пакеты и помечают, что для работы ПО, предоставляемого пакетом A, необходимо установить пакет B. В таком случае говорят, что пакет A **зависит** от пакета B или что между пакетами A и B существует **зависимость**.

Отслеживание зависимостей между такими пакетами представляет собой серьёзную задачу для любого дистрибутива — некоторые компоненты могут быть взаимозаменяемыми: может обнаружиться несколько пакетов, предлагающих затребованный ресурс.

Задача контроля целостности и непротиворечивости установленного в системе ПО ещё сложнее. Представим, что некие программы A и B требуют наличия в системе компоненты C версии 1.0. Обновление версии пакета A, требующее обновления компоненты C до новой, использующей новый интерфейс доступа, версии (скажем, до версии 2.0), влечёт за собой обязательное обновление и программы B.

Однако менеджеры пакетов оказались неспособны предотвратить все возможные коллизии при установке или удалении программ, а тем более эффективно устранить нарушения целостности системы. Особенно сильно этот недостаток сказывается при обновлении систем из централизованного репозитория пакетов, в котором последние могут непрерывно обновляться, дробиться на более мелкие и т. п. Этот недостаток и стимулировал создание систем управления программными пакетами и поддержания целостности системы.

Для автоматизации этого процесса и применяется Усовершенствованная система управления программными пакетами APT (от англ. Advanced Packaging Tool). Такая автоматизация достигается созданием одного или нескольких внешних **репозиториев**, в которых хранятся пакеты программ и относительно которых производится сверка пакетов, установленных в системе. Репозитории могут содержать как официальную версию дистрибутива, обновляемую его разработчиками по мере выхода новых версий программ, так и локальные наработки, например, пакеты, разработанные внутри компании.

Таким образом, в распоряжении APT находятся две базы данных: одна описывает установленные в системе пакеты, вторая — внешний репозиторий. APT отслеживает целостность установленной системы и, в случае обнаружения противоречий в зависимостях пакетов, руководствуется сведениями о внешнем репозитории для разрешения конфликтов и поиска корректного пути их устранения.

Первоначально APT был разработан для управления установкой и удалением программ в дистрибутиве Debian GNU/Linux. При разработке ставилась задача заменить используемую в Debian систему выбора программных пакетов dselect на новую, обладающую большими возможностями и простым пользовательским интерфейсом, а также позволяющую производить установку, обновление и повседневные «хозяйственные» работы с установленными на машине программами без необходимости изучения тонкостей используемого в дистрибутиве менеджера программных пакетов.

Эти привлекательные возможности долгое время были доступны только пользователям Debian, поскольку в APT поддерживался только один менеджер пакетов, а именно применяемый в Debian менеджер пакетов `dpkg`, несовместимый с используемым в ALT Linux RPM. Эта несовместимость заключается прежде всего в различии используемых форматов данных (хотя существуют программы-конвертеры), но имеются и другие различия, обсуждение которых выходит за рамки изложения.

APT, однако, изначально проектировался как не зависящий от конкретного метода работы с установленными в системе пакетами, и эта особенность позволила разработчикам из бразильской компании Conectiva реализовать в нём поддержку менеджера пакетов RPM. Таким образом, пользователи основанных на RPM дистрибутивов (дистрибутивы ALT Linux входят в их число) получили возможность использовать этот мощный инструмент.

Система APT состоит из нескольких утилит. Чаще всего используется утилита управления пакетами `apt-get`: она автоматически определяет зависимости между пакетами и строго следит за их соблюдением при выполнении любой из следующих операций: установка, удаление или обновление пакетов.

Графический интерфейс для APT

APT — это пакет утилит для работы из командной строки. Для любителей средств управления с графическим интерфейсом доступно на выбор несколько графических оболочек для APT. Одна из самых удачных — synaptic, она значительно проще в использовании, чем другие оболочки для APT. Вместо использования дерева для отображения пакетов synaptic основан на мощной системе фильтрации пакетов. Это значительно упрощает интерфейс и вместе с тем предоставляет гораздо больше гибкости при навигации по очень длинным спискам пакетов. Если вы хотите устанавливать и удалять пакеты и предпочитаете работать с программами с графическим интерфейсом, прежде всего попробуйте synaptic.

В ALT Linux также разработана графическая оболочка для APT — alterator-packages, которая основана на платформе Alterator и используется при установке системы и в средстве настройки ALT Linux Control Center.

Источники программ (репозитории)

Репозитории, с которыми работает APT, отличаются от обычного набора пакетов наличием метаинформации — индексов пакетов, содержащихся в репозитории, и сведений о них. Поэтому, чтобы получить всю информацию о репозитории, APT достаточно получить его индексы.

APT может работать с любым количеством репозиториев одновременно, формируя единую информационную базу обо всех содержащихся в них пакетах. При установке пакетов APT обращает внимание только на название пакета, его версию и зависимости, а расположение в том или ином репозитории не имеет значения. Если потребуется, APT в рамках одной операции установки группы пакетов может пользоваться несколькими репозиториями.

APT позволяет взаимодействовать с репозиторием с помощью различных протоколов доступа. Наиболее популярные — HTTP и FTP, однако существуют и некоторые дополнительные методы.

Для того, чтобы APT мог использовать тот или иной репозиторий, информацию о нем необходимо поместить в файл

`/etc/apt/sources.list`[1]. Описания репозиториев заносятся в этот файл в следующем виде:

```
rpm [подпись] метод:путь база название
rpm-src [подпись] метод:путь база название
```

`rpm или rpm-src`

Тип репозитория (скомпилированные программы или исходные тексты).

`подпись`

Необязательная строка-указатель на электронную подпись разработчиков. Наличие этого поля подразумевает, что каждый пакет из данного репозитория должен быть подписан соответствующей электронной подписью. Подписи описываются в файле `/etc/apt/vendor.list`.

`метод`

Способ доступа к репозиторию: `ftp`, `http`, `file`, `rsh`, `ssh`, `cdrom`.

`путь`

Путь к репозиторию в терминах выбранного метода.

`база`

Относительный путь к базе данных репозитория.

`название`

Название репозитория.

Для добавления в `sources.list` репозитория на компакт-диске в APT даже предусмотрена специальная утилита — `apt-cdrom`. Чтобы добавить запись о репозитории на компакт-диске, достаточно вставить диск в привод и выполнить команду `apt-cdrom add`. После этого в `sources.list` появится запись о подключённом диске примерно такого вида:

```
rpm cdrom:[Junior Disk 1]/ ALTLinux main
rpm-src        cdrom:[Junior Disk 1]/ ALTLinux main
```

[1] Если быть точным, этот файл может называться и иначе, другое имя файла со списком источников программ можно указать в конфигурационном файле `/etc/apt/apt.conf`. Подробнее о формате файла `apt.conf` можно узнать из руководства apt.conf(5).

После того как отредактирован список репозиториев в `sources.list`, необходимо обновить локальную базу данных APT о доступных пакетах. Это делается командой `apt-get update`.

Если в `sources.list` присутствует репозиторий, содержимое которого может изменяться (как происходит с любым постоянно разрабатываемым репозиторием, в частности, Sisyphus[1]), то прежде чем работать с APT, необходимо синхронизировать локальную базу данных с удалённым сервером командой `apt-get update`. Локальная база данных создаётся заново каждый раз, когда в репозитории происходит изменение: добавление, удаление или переименование пакета. Для репозиториев, находящихся на компакт-дисках и подключённых командой `apt-cdrom add`, синхронизацию достаточно сделать один раз в момент подключения.

При выборе пакетов для установки, APT руководствуется *всеми* доступными репозиториями вне зависимости от способа доступа к ним. Так, если в репозитории, доступном по сети Интернет, обнаружена более новая версия программы, чем на компакт-диске, то APT начнёт загружать данный пакет из Интернет. Поэтому если подключение к Интернет отсутствует или ограничено низкой пропускной способностью канала или выской стоимостью, то следует закомментировать те строчки в `/etc/apt/sources.list`, в которых говорится о ресурсах, доступных по Интернет.

Репозитории ALT Linux

Все дистрибутивы ALT Linux выпускаются на основе репозитория Sisyphus команды ALT Linux Team[2]. Следует иметь в виду, что Sisyphus не является самостоятельным дистрибутивом, а отражает текущее состояние разработки и может содержать нестабильные версии пакетов. Периодически на базе этого проекта выпускаются отдельные оттестированные «срезы» — дистрибутивы.

В отличие от Sisyphus, ежедневно обновляемого разработчиками, такие срезы являются «замороженными» — разработка в них не ведётся, и сами срезы сохраняются в целях обеспечения целостности среды дистрибутива, в которой уже не должны обновляться версии пакетов. Единственное исключение делается для обновлений, исправляющих проблемы в безопасности системы, однако такие обновления помещаются в отдельном репозитории для каждого дистрибутива. Срезы Sisyphus и репозитории обновлений также являются полноценными репозиториями APT.

[1] `http://sisyphus.ru`
[2] `http://www.altlinux.ru`

Непосредственно после установки дистрибутива ALT Linux в `/etc/apt/sources.list` обычно указывается несколько репозиториев:

- репозиторий обновлений в системе безопасности дистрибутива;
- полный срез репозитория Sisyphus, подмножеством которого является дистрибутив.

Поиск пакетов

Если вы не знаете точного названия пакета, для его поиска можно воспользоваться утилитой `apt-cache`, которая позволяет искать не только по имени пакета, но и по его описанию.

Команда `apt-cache search подстрока` позволяет найти все пакеты, в именах или описании которых присутствует указанная подстрока. Например:

```
$ apt-cache search master
xcdroast - A GUI program for burning Cds
bluefish - A WYSIWYG GPLized HTML editor
xmess - X-Mess Multi Emulator Super System
mkisofs - Creates an image of an ISO9660 filesystem
```

Для того, чтобы подробнее узнать о каждом из найденных пакетов и прочитать его описание, можно воспользоваться командой `apt-cache show`, которая покажет информацию о пакете из репозитория:

```
Пакет: bluefish
Секция: Networking/WWW
Размер установленных пакетов: 2018
Упаковщик: AEN <aen@logic.ru>
Версия: 1:0.7-alt0.1
..
Предоставляет: bluefish
Архитектура: i586
..
Имя файла: bluefish-0.7-alt0.1.i586.rpm
Описание: A WYSIWYG GPLized HTML editor
Bluefish is a programmer's HTML editor, designed to save the
experienced webmaster some keystrokes. It features a multiple
```

```
file editor, multiple toolbars, custom menus, image and thumbnail
dialogs, open from the web, HTML validation and lots of wizards.
It is in continuous development, but it's already one of the best
WYSIWYG HTML editors.
```

Как можно заметить, в кратком описании этого пакета нет слова «master», которое было задано в качестве подстроки для поиска. Однако здесь присутствует слово «webmaster», что объясняет наличие этого пакета в результате поиска по слову «master».

`apt-cache` позволяет осуществлять поиск и по русскому слову, однако в этом случае будут найдены только те пакеты, у которых помимо английского есть ещё и описание на русском языке. К сожалению, русское описание на настоящий момент есть не у всех пакетов, хотя описания наиболее актуальных для пользователя пакетов переведены.

Установка или обновление пакета

Установка пакета с помощью APT выполняется командой

```
# apt-get install имя_пакета
```

`apt-get` позволяет устанавливать в систему пакеты, требующие для работы другие, пока ещё не установленные. В этом случае он определяет, какие пакеты необходимо установить, и устанавливает их, пользуясь всеми доступными репозиториями.

Установка пакета `clanbomber` командой `apt-get install clanbomber` приведёт к следующему диалогу с APT:

```
Обработка файловых зависимостей... Завершено
Чтение списков пакетов... Завершено
Построение дерева зависимостей... Завершено
Следующие дополнительные пакеты будут установлены:
clanlib clanlib-mikmod clanlib-sound libmikmod
Следующие НОВЫЕ пакеты будут установлены:
clanbomber clanlib clanlib-mikmod clanlib-sound libmikmod
0 пакетов будет обновлено, 5 будет добавлено новых,
0 будет удалено(заменено) и 0 не будет обновлено.
Необходимо получить 0B/2577kB архивов. После распаковки 3862kБ
будет
использовано.
Продолжить? [Y/n] y
Выполняется программа RPM (/bin/rpm -Uv --replacepkgs -h)...
Подготовка... ########################################
```



```
libmikmod ############################################
clanlib ##############################################
clanlib-mikmod #######################################
clanlib-sound ########################################
clanbomber ###########################################
```

> `apt-get` всегда запрашивает подтверждение на выполнение операции установки и обновления, за исключением случая, когда требуется установить (или обновить) только один пакет. Если вы не уверены в том, что в результате выполнения операции система останется работоспособной, запустите `apt-get` с опцией `-S`, в этом случае будет показан отчёт о выполнении операции обновления, но в действительности обновление произведено не будет.

Команда `apt-get install имя_пакета` используется и для обновления уже установленного пакета или группы пакетов. В этом случае `apt-get` дополнительно проверяет, не обновилась ли версия пакета в репозитории по сравнению с установленным в системе.

При помощи APT можно установить и отдельный бинарный rpm-пакет, не входящий ни в один из репозиториев (например, полученный из Интернет). Для этого достаточно выполнить команду `apt-get install путь_к_файлу`. При этом APT проведёт стандартную процедуру проверки зависимостей и конфликтов с уже установленными пакетами.

Иногда, в результате операций с пакетами без использования APT, целостность системы нарушается, и `apt-get` отказывается выполнять операции установки, удаления или обновления. В этом случае необходимо повторить операцию, задав опцию `-f`, заставляющую `apt-get` исправить нарушенные зависимости, удалить или заменить конфликтующие пакеты. В этом случае необходимо внимательно следить за сообщениями, выдаваемыми `apt-get`. Любые действия в этом режиме обязательно требуют подтверждения со стороны пользователя.

Удаление установленного пакета

Для удаления пакета используется команда `apt-get remove имя_пакета`. Для того, чтобы не нарушать целостность системы, будут удалены и все пакеты, зависящие от удаляемого: если отсутствует необходимый для работы приложения компонент (например, библиотека), то само приложение становится бесполезным. В случае удаления пакета,

который относится к базовым компонентам системы, apt-get потребует дополнительного подтверждения производимой операции с целью предотвратить возможную случайную ошибку.

Если вы попробуете при помощи apt-get удалить базовый компонент системы, вы увидите такой запрос на подтверждение операции:

```
# apt-get remove filesystem
Обработка файловых зависимостей... Завершено
Чтение списков пакетов... Завершено
Построение дерева зависимостей... Завершено
Следующие пакеты будут УДАЛЕНЫ:
basesystem filesystem ppp sudo
Внимание: следующие базовые пакеты будут удалены:
В обычных условиях этого не должно было произойти, надеемся, вы
точно
представляете, чего требуете!
basesystem filesystem (по причине basesystem)
0 пакетов будет обновлено, 0 будет добавлено новых, 4 будет
удалено(заменено) и 0 не будет обновлено.
Необходимо получить 0B архивов. После распаковки 588kБ будет
освобождено.
Вы собираетесь совершить потенциально вредоносное действие
Для продолжения, наберите по-английски
'Yes, I understand this may be bad'
(Да, я понимаю, что это может быть плохо).
```

Каждую ситуацию, в которой APT выдаёт такое сообщение, необходимо рассматривать отдельно. Однако, вероятность того, что после выполнения этой команды система окажется неработоспособной, очень велика.

Обновление всех установленных пакетов

Для обновления всех установленных пакетов используется команда apt-get upgrade. Она позволяет обновить те и только те установленные пакеты, для которых в репозиториях, перечисленных в /etc/apt/sources.list, имеются новые версии; при этом из системы не будут удалены никакие другие пакеты. Этот способ полезен при работе со стабильными пакетами приложений, относительно которых известно, что они при смене версии изменяются несущественно.

Иногда, однако, происходит изменение в именовании пакетов или изменение их зависимостей. Такие ситуации не обрабатываются ко-

мандой apt-get upgrade, в результате чего происходит нарушение целостности системы: появляются неудовлетворённые зависимости. Например, переименование пакета MySQL-shared, содержащего динамически загружаемые библиотеки для работы с СУБД MySQL, в libMySQL (отражающая общую тенденцию к наименованию библиотек в дистрибутиве) не приводит к тому, что установка обновлённой версии libMySQL требует удаления старой версии MySQL-shared. Для разрешения этой проблемы существует режим обновления в масштабе дистрибутива — apt-get dist-upgrade.

В случае обновления всего дистрибутива APT проведёт сравнение системы с репозиторием и удалит устаревшие пакеты, установит новые версии присутствующих в системе пакетов, а также отследит ситуации с переименованиями пакетов или изменения зависимостей между старыми и новыми версиями программ. Всё, что потребуется поставить (или удалить) дополнительно к уже имеющемуся в системе, будет указано в отчёте apt-get, которым APT предварит само обновление.

При работе с Sisyphus для обновления системы рекомендуется использовать команду apt-get dist-upgrade.

См. также Alterator-packages: графическое средство управления пакетами . . . . . [снаружи, стр. 71]

Hasher: безопасная технология сборки пакетов [стр. 108]

Как получить нужную программу

Кирилл Маслинский

Предположим, что вы уже точно определились со своими потребностями и выяснили тем или иным способом название «линуксовой» программы, которая вам нужна. Как её раздобыть и установить?

Поиск среди установленных пакетов

Прежде всего, проверьте — может быть, нужная вам программа уже установлена. Обычно в качестве названия необходимой «программы» удаётся выяснить не название конкретного исполняемого файла, а название проекта, в рамках которого эта программа разрабатывается

(хотя они нередко совпадают или как минимум похожи). Почти наверняка название проекта будет совпадать или составлять часть имени соответствующего **пакета** в ALT Linux. Для примера представим, что мы заинтересовались (прочитав в рассылке) известным почтовым клиентом для Linux — mutt, который часто упоминается с эпитетами «мощный» и «гибкий».

Проверим, нет ли среди установленных пакета с похожим именем:

```
tester@tacit:~$ rpm -qa | grep -i mutt
mutt-1.4.2.1i-alt5
```

Команда `rpm -qa` выводит список всех пакетов, установленных в системе, затем этот список был передан фильтру `grep` для поиска нужной строки, специально вызванному с ключом `-i`, чтобы возможные различия в прописных/строчных буквах в названии не помешали найти нужное. В результате найдено полное (включающее номер версии и сборки) имя установленного пакета.

Итак, пакет `mutt` уже установлен, теперь нужно узнать, как с ним обращаться — для этого полезно выяснить, какая в этом пакете имеется документация.

```
tester@tacit:~$ rpm -qld mutt
/usr/share/doc/mutt-1.4.2.1i/COPYRIGHT
/usr/share/doc/mutt-1.4.2.1i/ChangeLog
. . .
/usr/share/man/man1/flea.1.gz
/usr/share/man/man1/mutt.1.gz
/usr/share/man/man1/muttbug.1.gz
/usr/share/man/man5/mbox.5.gz
/usr/share/man/man5/muttrc.5.bz2
```

Список выводится довольно длинный (на один экран не вмещается), так что наверняка можно разобраться во всех тонкостях этого почтового клиента, внимательно почитав документацию. В конце списка обнаруживается несколько руководств, которые можно просмотреть командой `man`.

К сожалению, не во всех пакетах дела с документацией обстоят так же хорошо, как в `mutt`. Иногда дело ограничивается коротеньким README, иногда нет и того. В таком плачевном случае остаётся выяснить, какие в пакете есть утилиты (их может быть и несколько, и необязательно их названия будут напоминать имя пакета), и попробовать добиться от них краткой справки по использованию, вызвав с ключом `--help`.


```
tester@tacit:~$ rpm -ql mutt | grep bin
/usr/bin/flea
/usr/bin/mutt
/usr/bin/muttbug
/usr/bin/pgpewrap
/usr/bin/pgpring
```

Команда `rpm -ql имя_пакета` выводит список всех файлов, которые принадлежат указанному пакету. Это работает только для уже установленных пакетов, и, соответственно, уже существующих в системе файлов. *Имя_пакета* можно указывать без номера версии, как в нашем примере. Профильтровав список файлов в поисках строки `bin` — названия каталога для исполняемых файлов — мы получили список всех утилит в этом пакете.

Дистрибутив

Не все из входящих в дистрибутив пакетов (а это обычно несколько тысяч) включаются в установку по умолчанию. Для поиска среди имеющихся в дистрибутиве, но ещё не установленных пакетов можно воспользоваться менеджером пакетов APT (команда `apt-cache search`) или любой графической оболочкой к нему, например, alterator-packages или synaptics. В дистрибутиве APT обычно настроен таким образом, что сразу после установки в кеше APT хранится база сведений обо всех вошедших в дистрибутив пакетах, даже если они разложены по нескольким компакт-дискам. Установить найденный пакет можно стандартными средствами `apt-get` или названных графических оболочек.

Contrib

В дистрибутив попадают только специально *отобранные* пакеты, которые проходят тестирование и, возможно, дополнительную настройку, чтобы представить пользователю уже заранее приспособленную для определённых задач *систему*. В дистрибутив нельзя включить произвольную программу из произвольного источника, причина тому — **зависимости** между программами. Любая несогласованность в зависимостях (например, более старая или более новая версия системной библиотеки) может привести к неработоспособности программы или даже к невозможности её установить. Поэтому дистрибутив составляется не просто из множества пакетов, а из целостного

репозитория, в котором строго согласованы версии всех пакетов. Та часть этого репозитория, которая не вошла в дистрибутив, называется несколько жаргонным, но очень точным полусловом **contrib**[1].

Именно в contrib остаются те пакеты, которым не посчастливилось войти в дистрибутив. И именно в contrib нужно искать в первую очередь потому, что если программа найдётся там, то она гарантированно установится в вашей системе. А если ей потребуется что-то ещё для установки, то гарантированно найдутся все нужные пакеты нужных версий либо в дистрибутиве, либо в том же contrib.

Contrib может распространяться вместе или параллельно с дистрибутивом на компакт-дисках. Так, contrib к дистрибутиву ALT Linux 3.0 Compact включён в DVD-версию дистрибутива. Кроме того, полные репозитории, включающие contrib, к этому и другим дистрибутивам доступны на официальном ftp-сервере ALT Linux[2] и его зеркалах.

Для работы с contrib следует воспользоваться всё тем же стандартным менеджером пакетов APT. Единственное, что для этого требуется — указать соответствующий репозиторий в списке источников APT (указать URL в `/etc/apt/sources.list` или выполнить `apt-cdrom add`, подробности см. в документации по APT). После этого можно работать с пакетной базой дистрибутива и contrib как с единым целым, никаких противоречий и конфликтов не возникнет.

В погоне за новым

Иногда нужная программа в дистрибутиве есть, но уже достаточно старая, так что какие-то необходимые возможности появились после того, как вышел дистрибутив, и хочется программу обновить. Бывают ли «обновления» и где их взять?

Updates

Для каждого дистрибутива в обязательном порядке выпускаются и публикуются в Сети обновления. Например, для дистрибутива ALT Linux 3.0 Compact их можно найти на официальном ftp-сервере[3] и, естественно, на его зеркалах. Такие обновления делаются в том случае, если уже после выпуска дистрибутива в вошедших в него

[1] Полусловом — потому, что это не целое слово, а традиционно сложившееся сокращённое имя для соответствующего каталога на ftp.

[2] `ftp://ftp.altlinux.org/pub/distributions/ALTLinux/`

[3] `ftp://ftp.altlinux.ru/pub/distributions/ALTLinux/updates/3.0/`

программах обнаруживаются серьёзные ошибки, в первую очередь, угрожающие безопасности системы. Но и только: такие обновления не предназначены для включения новых возможностей (с новыми возможностями приходят новые ошибки!), поэтому регулярно делать такие обновления *крайне желательно*, а вот новой функциональности вы там не найдёте.

Строки для указания репозитория с обновлениями обычно присутствуют в `/etc/apt/sources.list` сразу после установки, но могут быть закомментированы. Вам потребуется только раскомментировать нужные (например, выбрав подходящее ftp-зеркало) и выполнить команду `apt-get update && apt-get upgrade`.

Backports

Новые версии программ для старого дистрибутива — это как раз та цель, которую преследуют создатели репозитория backports[1]. Однако тут никто не может дать гарантий, что новые версии будут работать так же стабильно, как и старые. Работать с этими репозиториями следует так же, как и с updates, указав в их списке источников APT, конкретные инструкции приведены на сайте репозитория, там же публикуются новости об обновлённых версиях программ.

Sisyphus

Откуда же берутся новые версии? Теперь уже никак нельзя умолчать о том главном ежедневно обновляемом репозитории, срезы которого становятся основой для отбора пакетов в дистрибутивы ALT Linux — Sisyphus[2]. Именно в этом репозитории разработчики периодически публикуют новые версии своих пакетов (а в пакетах — новые версии программ), и весь процесс его разработки олицетворяет постоянное совершествование программ[3].

В Sisyphus включены конечно же, не все существующие свободные программы для Linux, но очень и очень многие — сейчас в нём более пяти тысяч исходных пакетов. Регулярно в нём появляются новые пакеты, ранее не собиравшиеся, но также регулярно удаляются некоторые из старых, которые потеряли актуальность, интерес разработчиков или работоспособность (или даже всё это вместе).

[1] `http://backports.altlinux.ru`

[2] `http://sisyphus.ru`

[3] Недаром репозиторий назван в честь труженика Сизифа.

Sisyphus может служить чем-то вроде рекомендательного списка для выбора среди невероятного количества альтернатив, предлагаемых миром свободных программ. Чем больше программа используется, чем она перспективнее, надёжнее, эффективнее — тем больше вероятность, что в ALT Linux Team найдётся тот, кто соберёт её для Sisyphus. Для пользователей дистрибутивов ALT Linux Sisyphus — это тот источник, о котором не нужно забывать, если программа не нашлась ни в дистрибутиве, ни в обновлениях. Информацию о переходе на Sisyphus с конкретных дистрибутивов ALT Linux можно найти на сайте разработчиков[1].

Не спешите включать его в список источников APT. Sisyphus — репозиторий заведомо нестабильный. Самое новое — не значит самое проверенное и гарантированно работающее!

Если требуется «точечное» обновление конкретного пакета, подходящая версия которого имеется в Sisyphus, то самый разумный подход — написать в специальный список рассылки[2] и предложить собрать этот пакет для вашего дистрибутива. Не исключено, что вам придётся сделать это и самостоятельно, но не стоит этого бояться — благодаря технологии hasher, используемой в ALT Linux для сборки пакетов, пересобрать пакет из Sisyphus для backports можно за несколько стандартных шагов. Для пересборки потребуется только исходный пакет из Sisyphus и полный репозиторий (contrib) соответствующего дистрибутива.

Самостоятельная сборка из исходных текстов

Когда б вы знали, из какого сора...

Широко распространено представление, что «по-настоящему крутые линуксоиды» сами должны компилировать («собирать») программы из исходных текстов. В нашем изложении эта возможность не случайно оказалась на последнем месте: если сработал хоть один из вышеназванных способов — собирать программу из исходных текстов не следует. В общем случае, даже если единственный способ доступа к репозиторию для вас — это модем, всё равно выйдет быстрее и

[1] `http://wiki.sisyphus.ru`

[2] `http://lists.altlinux.org/mailman/listinfo/backports`

удобнее использовать готовую пакетную базу, чем самостоятельно собирать уже кем-то собранные программы.

Тому можно назвать огромную массу технических и нетехнических причин: хотя бы то, что в репозитории Sisyphus приняты весьма высокие стандарты на качество сборки, и поэтому если она там есть, то наверняка на достаточно профессиональном уровне. В сборке часто оказывается множество подводных камней, которые могут сделать этот процесс долгим, мучительным и безрезультатным. А следы даже безрезультатной сборки, проведённой без должных предосторожностей прямо в рабочей системе, могут эту самую систему повредить до полной неработоспособности.

И не всегда следы неудачной установки так легко стереть. Представьте себе ситуацию, когда для сборки программа устанавливает в системе свою собственную версию стандартной системной библиотеки (если собирать от имени суперпользователя, это пройдёт гладко и незаметно). А если сборочная процедура взятого из непроверенного источника ПО содержит злонамеренные фрагменты, которые вы собственной рукой запустите с правами суперпользователя?

После перечисленных ужасов может возникнуть вопрос: но ведь не боги программы собирают? Именно так, и если вы в состоянии сделать это грамотно (или готовы научиться) и обойти все подводные камни (или готовы наткнуться) — отчего же не поделиться результатами своего труда с другими? Соберите пакет для Sisyphus![1]

См. также
- Дистрибутив ALT Linux 3.0 Compact [снаружи, стр. 12]
- Подробнее о зависимостях . . . . . [стр. 91]
- В примерах использована фильтрация вывода команд с помощью `grep` — конвейер . . . . . [стр. 112]

[1] Подробности — на `http://wiki.sisyphus.ru`.

Сборка программ для ALT Linux с использованием hasher

Александр Колотов

Введение

Часто во время сборки пакетов по зависимостям предлагается установить очень много «лишнего» ПО. Это не всегда бывает удобно: не хочется «мусорить» в уже рабочей системе или вообще нежелательно в ней иметь некоторые средства (например, средства разработки на сервере). Часто банальной причиной является нежелание где-то искать это ПО — этот процесс утомляет, особенно если репозиторий пакетов для дистрибутива представлен в виде нескольких дисков, то всё сводится к «жонглированию» дисками. . .

Поэтому в дистрибутивы ALT Linux попадают только пакеты, заботливой рукой мантейнера проведённые через технологию hasher. Эта технология позволяет собирать пакеты внутри «чистого» окружения (chroot), которое создаётся из числа пакетов, входящих в состав описанных на пользовательской машине репозиториев. Т.е. в чистом окружении устанавливаются все пакеты, необходимые для сборки данного (сборочная среда), затем в этом окружении и происходит сборка. В реальную систему пакеты при этом не ставятся. Таким образом, на входе hasher получает src.rpm пакет, а на выходе выдаёт уже собранные (только из данного src.rpm) пакеты, которые тут же можно использовать в наших репозиториях для установки на рабочие системы.

За технологию hasher отвечают два пакета: hasher и hasher-priv, для работы последнего также нужен пакет sisyphus_check. Внутри пакета hasher содержится краткая, но ёмкая документация: README, QUICKSTART и FAQ файлы, а также man-страницы, правда, они на английском языке, хотя разработчики программы вполне русскоговорящие. Этой документации, в принципе, хватает, чтобы начать работать с hasher, а в FAQ рассмотрены некоторые проблемы, которые могут возникать в работе. Но рассмотрим дополнительно некоторые аспекты работы с hasher.

Установка hasher

На данном этапе все просто. От пользователя root выполняем: `apt-get install hasher`. Затем необходимо подготовить hasher — объявить, что пользователь имеет право пользоваться услугами hasher: `hasher-useradd имя_пользователя`.

Теперь этому пользователю нужно заново войти в систему (причём предварительно полностью завершив сеанс, запуск нового терминального сеанса, например, в konsole, не приведёт к нужному эффекту).

Сборка программ в hasher

Сейчас работа должна происходить от обычного пользователя (но добавленного с помощью hasher-useradd). Сначала необходимо создать каталог, в котором будет строиться сборочная среда: `mkdir HASHER`.

Название и местоположение рабочего каталога hasher могут быть произвольными, но он должен быть доступен на запись пользователю, запускающему сборку. Кроме того, его не следует располагать на файловой системе, которая смонтирована с опциями `noexec` или `nodev`. Следует также обратить внимание, что hasher лучше работает с локальными репозиториями, то есть описанными (в `/etc/apt/sources.list`) с помощью `file:` или `copy:`. Хотя при достаточно новом APT не должно возникнуть проблем и с другими видами доступа.

После этого можно приступать к сборке:

```
$ hsh HASHER freetype-2.1.9-alt2.src.rpm
```

здесь

- `HASHER` — рабочий каталог для создания сборочной среды;
- `freetype-2.1.9-alt2.src.rpm` — пакет, который необходимо собрать.

При удачной сборке полученные пакеты будут лежать в `HASHER/repo/платформа/RPMS.hasher/`. А если произошла ошибка, то информация о ней будет выведена на экран. Например, это информация о том, что какие-то пакеты, необходимые для сборки данного, не были обнаружены в доступных репозиториях.

Кстати, указанный выше репозиторий (`HASHER/repo/платформа/RPMS.hasher/`) можно использовать для установки, указав его наравне с прочими источниками в `/etc/apt/sources.list`.

Многократная сборка[1]

```
$ mkdir build
$ hsh --initroot-only build
$ hsh-install <list buildrequires>
$ cp some.rpm build/chroot/.in
$ hsh-shell build
$ rpm -i some.rpm
$ cd /usr/src/RPM/SPECS
$ rpm -ba some.spec
```

Самостоятельная сборка пакетов

Следующая проблема может возникнуть при самосборных src.rpm: hasher будет выдавать сообщение о неверном поле `Packager` у собираемого пакета:

```
some-packet.src.rpm: wrong Packager: Nekto Sleva <nekto@gnu.org>
```

Тут всё дело в том, что hasher предназначается для тестирования сборки пакетов перед отправкой их в Sisyphus, поэтому помимо самой сборки происходит проверка на правильность заполнения служебной информации об rpm-пакете. В случае с Sisyphus, пакеты в нём могут размещать только члены ALT Linux Team, соответственно, в поле `Packager` должен быть указан один их них. Это проверяется утилитой sisyphus_check по наличию `altlinux.com|net|org|ru` после «`@`». `man hsh` говорит, что данную проверку можно отключить, указав опцию `--no-sisyphus-check[=LIST]`, где `LIST` — список пропускаемых тестов. О тестах, которые может пропустить sisyphus_check, логичнее спросить у него самого: `sisyphus_check --help`.

Итак, пробуем отключить тест на `Packager`, а заодно и проверку GPG-подписи:

```
$ hsh --target=i586 --no-sisyphus-check=packager,gpg HASHER
some-packet.src.rpm
```

Теперь работает.[2]

[1] Описал Женя Остапец.

[2] Спасибо за помощь в «разборках» с hasher Алексею Фролову aka raorn.

Часть III

Конструктор

Глава 7

Командная строка

Ввод, вывод и конвейер

Кирилл Маслинский, Мэтт Уэлш (Matt Welsh) и другие

Стандартный ввод и стандартный вывод

Программа обычно ценна тем, что может обрабатывать данные: принимать одно, на выходе выдавать другое, причём в качестве данных может выступать практически что угодно: текст, числа, звук, видео... Потоки входных и выходных данных для команды называются **ввод** и **вывод**. Потоков ввода и вывода у каждой программы может быть и по несколько. В Linux каждый процесс при создании в обязательном порядке получает так называемые **стандартный ввод** (standard input, stdin), **стандартный вывод** (standard output, stdout) и **стандартный вывод ошибок** (standard error, stderr).

Стандартные потоки ввода/вывода предназначены в первую очередь для обмена текстовой информацией. Тут даже не важно, кто общается с помощью текстов: человек с программой или программы между собой — главное, чтобы у них был канал передачи данных и чтобы они говорили «на одном языке».

Текстовый принцип работы с машиной позволяет отвлечься от конкретных частей компьютера, вроде системной клавиатуры и видеокарты с монитором, рассматривая единое *оконечное устройство*, посредством которого пользователь вводит текст (команды) и передаёт его системе, а система выводит необходимые пользователю данные и сообщения (диагностику и ошибки). Такое устройство называется **терминалом**. В общем случае терминал — это точка входа пользователя в систему, обладающая способностью передавать текстовую информацию. Терминалом может быть отдельное внешнее устройство, подключаемое к компьютеру через порт последовательной передачи данных (в персональном компьютере он называется «COM port»). В роли терминала может работать (с некоторой поддержкой со стороны системы) и программа (например, xterm или ssh). Наконец, **виртуальные консоли** Linux — тоже терминалы, только организованные программно с помощью подходящих устройств современного компьютера.

При работе с командной строкой, стандартный ввод командной оболочки связан с клавиатурой, а стандартный вывод и вывод ошибок — с экраном монитора (или окном эмулятора терминала). Покажем на примере простейшей команды — cat. Обычно команда `cat` читает данные из всех файлов, которые указаны в качестве её параметров, и посылает считанное непосредственно в стандартный вывод (stdout). Следовательно, команда

```
/home/larry/papers# cat history-final masters-thesis
```

выведет на экран сначала содержимое файла `history-final`, а затем — файла `masters-thesis`.

Однако если имя файла не указано, программа `cat` читает входные данные из stdin и немедленно возвращает их в stdout (никак не изменяя). Данные проходят через `cat`, как через трубу. Приведём пример:

```
/home/larry/papers# cat
Hello there.
Hello there.
Bye.
Bye.
Ctrl+D
/home/larry/papers#
```

Каждую строчку, вводимую с клавиатуры, программа cat немедленно возвращает на экран. При вводе информации со стандартного ввода конец текста сигнализируется вводом специальной комбинации клавиш, как правило *Ctrl–D*.

Приведём другой пример. Команда `sort` читает строки вводимого текста (также из stdin, если не указано ни одного имени файла) и выдаёт набор этих строк в упорядоченном виде на stdout. Проверим её действие.

```
/home/larry/papers# sort
bananas
carrots
apples
Ctrl+D
apples
bananas
carrots
/home/larry/papers#
```

Как видно, после нажатия *Ctrl–D*, `sort` вывела строки упорядоченными в алфавитном порядке.

Перенаправление ввода и вывода

Допустим, вы хотите направить вывод команды sort в некоторый файл, чтобы сохранить упорядоченный по алфавиту список на диске. Командная оболочка позволяет перенаправить стандартный вывод команды в файл, используя символ «>». Приведём пример:

```
/home/larry/papers# sort > shopping-list
bananas
carrots
apples
Ctrl+D
/home/larry/papers#
```

Можно увидеть, что результат работы команды sort не выводится на экран, однако он сохраняется в файле с именем `shopping-list`. Выведем на экран содержимое этого файла:

```
/home/larry/papers# cat shopping-list
apples
bananas
carrots
/home/larry/papers#
```

Пусть теперь исходный неупорядоченный список находится в файле `items`. Этот список можно упорядочить с помощью команды `sort`, если указать ей, что она должна читать из данного файла, а не из своего стандартного ввода, и кроме того, перенаправить стандартный вывод в файл, как это делалось выше. Пример:


```
/home/larry/papers# sort items shopping-list
/home/larry/papers# cat shopping-list
apples
bananas
carrots
/home/larry/papers#
```

Однако можно поступить иначе, перенаправив не только стандартный вывод, но и *стандартный ввод* утилиты из файла, используя для этого символ «<»:

```
/home/larry/papers# sort < items
apples
bananas
carrots
/home/larry/papers#
```

Результат команды `sort < items` эквивалентен команде `sort items`, однако первая из них демонстрирует следующее: при выдаче команды `sort < items` система ведёт себя так, как если бы данные, которые содержатся в файле `items`, были введены со стандартного ввода. Перенаправление осуществляется командной оболочкой. Команде `sort` не сообщалось имя файла `items`: эта команда читала данные из своего стандартного ввода, как если бы мы вводили их с клавиатуры.

Введём понятие **фильтра**. Фильтром является программа, которая читает данные из стандартного ввода, некоторым образом их обрабатывает и результат направляет на стандартный вывод. Когда применяется перенаправление, в качестве стандартного ввода и вывода могут выступать файлы. Как указывалось выше, по умолчанию, stdin и stdout относятся к клавиатуре и к экрану соответственно. Программа sort является простым фильтром — она сортирует входные данные и посылает результат на стандартный вывод. Совсем простым фильтром является программа `cat` — она ничего не делает с входными данными, а просто пересылает их на выход.

Использование состыкованных команд (конвейер)

Выше уже демонстрировалось, как использовать программу sort в качестве фильтра. В этих примерах предполагалось, что исходные данные находятся в некотором файле или что эти исходные данные будут введены с клавиатуры (стандартного ввода). Однако как поступить,

если вы хотите отсортировать данные, которые являются результатом работы какой-либо другой команды, например, `ls`?

Будем сортировать данные в обратном алфавитном порядке; это делается опцией `-r` команды `sort`. Если вы хотите перечислить файлы в текущем каталоге в обратном алфавитном порядке, один из способов сделать это будет таким. Применим сначала команду `ls`:

```
/home/larry/papers# ls
english-list
history-final
masters-thesis
notes
/home/larry/papers#
```

Теперь перенаправляем выход команды `ls` в файл с именем `file-list`:

```
/home/larry/papers# ls > file-list
/home/larry/papers# sort -r file-list
notes
masters-thesis
history-final
english-list
/home/larry/papers#
```

Здесь выход команды `ls` сохранен в файле, а после этого этот файл был обработан командой `sort -r`. Однако этот путь является неизящным и требует использования временного файла для хранения выходных данных программы `ls`.

Решением в данной ситуации может служить создание **состыкованных команд** (pipelines). Стыковку осуществляет командная оболочка, которая stdout первой команды направляет на stdin второй команды. В данном случае мы хотим направить stdout команды `ls` на stdin команды `sort`. Для стыковки используется символ «|», как это показано в следующем примере:

```
/home/larry/papers# ls | sort -r
notes
masters-thesis
history-final
english-list
/home/larry/papers#
```


Эта команда короче, чем совокупность команд, и её проще набирать. Рассмотрим другой полезный пример. Команда

```
/home/larry/papers# ls /usr/bin
```

выдаёт длинный список файлов. Большая часть этого списка пролетает по экрану слишком быстро, чтобы содержимое этого списка можно было прочитать. Попробуем использовать команду more для того, чтобы выводить этот список частями:

```
/home/larry/papers# ls /usr/bin | more
```

Теперь можно этот список «перелистывать».

Можно пойти дальше и состыковать более двух команд. Рассмотрим команду `head`, которая является фильтром следующего свойства: она выводит первые строки из входного потока (в нашем случае на вход будет подан выход от нескольких состыкованных команд). Если мы хотим вывести на экран последнее по алфавиту имя файла в текущем каталоге, можно использовать следующую длинную команду:

```
/home/larry/papers# ls | sort -r | head -1 notes
/home/larry/papers#
```

где команда `head -1` выводит на экран первую строку получаемого ей входного потока строк (в нашем случае поток состоит из данных от команды `ls`), отсортированных в обратном алфавитном порядке.

Недеструктивное перенаправление вывода

Эффект от использования символа «>» для перенаправления вывода файла является деструктивным; иными словами, команда

```
/home/larry/papers# ls > file-list
```

уничтожит содержимое файла `file-list`, если этот файл ранее существовал, и создаст на его месте новый файл. Если вместо этого перенаправление будет сделано с помощью символов «>>», то вывод будет приписан в конец указанного файла, при этом исходное содержимое файла не будет уничтожено. Например, команда

```
/home/larry/papers# ls >> file-list
```

приписывает вывод команды `ls` в конец файла `file-list`.

Следует иметь в виду, что перенаправление ввода и вывода и стыкование команд осуществляется командными оболочками, которые поддерживают использование символов «>», «>>» и «|». Сами команды не способны воспринимать и интерпретировать эти символы.

См. также Примеры эффективного использования конвейера в сценариях . . . . . [стр. 156]

Удобства shell: экономия движений

Георгий Курячий, Кирилл Маслинский

Редактирование ввода

Редактирование командной строки

Некоторое время поработав в Linux, понабирав команды в командной строке, приходишь к выводу, что в общении с оболочкой не помешают кое-какие удобства. Одно из таких удобств — возможность редактировать вводимую строку с помощью клавиши *Backspace* (удаление последнего символа), *Ctrl+W* (удаление слова) и *Ctrl+U* (удаление всей строки) — предоставляет сам терминал Linux. Эти команды работают для *любого* построчного ввода в терминале. Если по каким-то причинам в строчку на экране влез мусор, можно нажать *Ctrl+R* (**r**edraw) — система выведет в новой строке содержимое входного буфера.

Командная оболочка поддерживает некоторые базовые операции по редактированию командной строки, которых можно ожидать для любого текстового ввода. Речь идёт о клавишах *Стрелка влево* и *Стрелка вправо*, с помощью которых можно перемещать курсор по командной строке, и клавише *Del*, удаляющей символ *под* курсором, а не позади него. Помимо этого перемещаться в командной строке можно не только по одному сиволу вперёд и назад, но и по словам: команды `ESCF`/`ESCB` или `Alt+F`/`Alt+B` соответственно (от **f**orward и **b**ackward), работают также клавиши *Home* и *End*, или, что то же самое, *Ctrl+A* и *Ctrl+E*.

История команд

Bash располагает весьма мощным механизмом — возможностью работать с **историей команд**. Все команды, набранные пользователем, `bash` запоминает и позволяет обращаться к ним впоследствии. Для работы с историей команд используются клавиши со стрелками — вверх и вниз. По стрелке вверх (можно использовать и *Ctrl+P*, **p**revious), список поданных команд «прокручивается» от последней к первой, а по стрелке вниз (*Ctrl+N*, **n**ext) — обратно. Соответствующая команда отображается в командной строке как только что набранная, её можно отредактировать и подать оболочке (подгонять курсор к концу строки при этом не обязательно).

Если необходимо добыть из истории какую-то давнюю команду, проще не гонять список истории стрелками, а *поискать* в ней с помощью команды *Ctrl+R* (**r**everse search). При этом выводится подсказка специального вида («(reverse-i-search)»), подстрока поиска (окружённая символами ‘ и ’) и последняя из команд в истории, в которой эта подстрока присутствует:

```
[methody@localhost methody]$
 ^R | (reverse-i-search)‘’:
 i  | (reverse-i-search)‘i’: ls i
 n  | (reverse-i-search)‘in’: info
 f  | (reverse-i-search)‘inf’: info
 o  | (reverse-i-search)‘info’: info
 ^R | (reverse-i-search)‘info’: man info
 ^R | (reverse-i-search)‘info’: info "(bash.info.bz2)Commands For
History"
```

Пример представляет символы вводимые пользователем (в левой части до «|») и содержимое последней строки терминала. Это «кадры» работы с одной и той же строкой, показывающие, как она меняется при наборе. Набрав «info», пользователь продолжил поиск этой подстроки, повторяя *Ctrl+R* до тех пор, пока не наткнулся на нужную ему команду, содержащую подстроку «`info`». Осталось только передать её `bash` с помощью *Enter*.

Чтобы история команд могла сохраняться *между* сеансами работы пользователя, `bash` записывает её в файл `.bash_history`, находящийся в домашнем каталоге пользователя. Делается это в момент *завершения* оболочки: накопленная за время работы история дописывается в конец этого файла. При следующем запуске `bash` считывает `.bash_history` целиком. История хранится не вечно, количество запоминае-

мых команд в `.bash_history` ограничено (обычно 500 командами, но это можно и перенастроить).

Сокращения

Поиск по истории — удобное средство: длинную командную строку можно не набирать целиком, а выискать и использовать. Однако *давнюю* команду придётся добывать с помощью нескольких *Ctrl+R* — а можно и совсем не доискаться, если она уже выбыла оттуда. Для того, чтобы оперативно заменять длинные команды короткими, стоит воспользоваться **сокращениями** (aliases). В конфигурационных файлах командного интерпретатора пользователя обычно *уже* определено несколько сокращений, список которых можно посмотреть с помощью команды `alias` без параметров:

```
[methody@localhost methody]$ alias
 alias cd..='cd ..'
 alias cp='cp -i'
 alias l='ls -lapt'
 alias ll='ls -laptc'
 alias ls='ls --color=auto'
 alias md='mkdir'
 alias mv='mv -i'
 alias rd='rmdir'
 alias rm='rm -i'
```

Выяснилось, что по команде `ls` вместо *утилиты* `/bin/ls` `bash` запускает собственную команду-сокращение, превращающееся в команду `ls --color=auto`. *Повторно* появившуюся в команде подстроку «`ls`» интерпретатор уже не обрабатывает, во избежание вечного цикла. Например, команда `ls -al` превращается в результате в `ls --color=auto -al`. Точно так же любая команда, начинающаяся с `rm`, превращается в `rm -i` (**i**nteractive), в результате чего ни одно удаление не обходится без вопросов в стиле «rm: удалить обычный файл ‘`файл`’?». Избавиться от ненужного сокращения можно с помощью команды `unalias`.

Достраивание

Сокращения позволяют быстро набирать *команды*, однако никак не затрагивают имён *файлов*, которые чаще всего и оказываются параметрами этих команд. Бывает, что набранной строки — пути к файлу

и нескольких первых букв его имени — достаточно для *однозначного* указания на этот файл, потому что по введённому пути больше файлов, чьё имя начинается на эти буквы, просто нет. Чтобы не дописывать оставшиеся буквы в `bash` можно нажать клавишу *Tab*. И `bash` *сам* достроит имя файла до полного (снова воспользуемся методом «кадров»):

```
[methody@localhost methody]$ ls -al /bin/base
 Tab        | [methody@localhost methody]$ ls -al /bin/basename
-rwxr-xr-x  1 root root 12520 Июн  3 18:29 /bin/basename
[methody@localhost methody]$ base
 Tab        | [methody@localhost methody]$ basename
 Tab        | [methody@localhost methody]$ basename ex
 Tab        | [methody@localhost methody]$ basename examples/
 Tab        | [methody@localhost methody]$ basename
examples/sample-file
sample-file
```

Дальше — больше. Оказывается, и имя команды можно вводить не целиком: оболочка догадается достроить набираемое слово именно до команды, раз уж это слово стоит в начале командной строки. Таким образом, команда `basename examples/sample-file` была набрана за *восемь* нажатий клавиш («base» и четыре *Tab*)! Не потребовалось вводить начало имени файла в каталоге `examples`, потому что файл там был всего один.

Выполняя **достраивание** (completion), `bash` может вывести не всю строку, а только ту её часть, относительно которой у него нет сомнений. Если дальнейшее достраивание может пойти *несколькими* путями, то однократное нажатие *Tab* приведёт к тому, что `bash` растерянно пискнет[1], а повторное — к выводу *под* командной строкой списка всех возможных вариантов. В этом случае надо подсказать командной оболочке продолжение: дописать несколько символов, определяющих, по какому пути пойдёт достраивание, и снова нажать *Tab*.

Дополнения в `bash` находятся ещё не на самой вершине удобства и экономии нажатий на клавиши. Если в `bash` *несколько* типов достраивания (по именам файлов, по именам команд и т. п.), то в `zsh`

[1] Все терминалы должны уметь выдавать звуковой сигнал при *выводе* управляющего символа *Ctrl+G*. Для этого не нужно запускать никаких дополнительных программ: «настоящие» терминалы имеют встроенный динамик, а виртуальные консоли обычно пользуются системным («пищалкой»). В крайнем случае разрешается привлекать внимание пользователя другими способами: например, эмулятор терминала `screen` пишет в служебной строке «wuff-wuff» («гав-гав»).

их *сколько угодно*: существует способ запрограммировать любой алгоритм достраивания и задать шаблон командной строки, в которой именно этот способ будет применяться.

Генерация имён файлов

Достраивание очень удобно, когда цель пользователя — задать *один* конкретный файл в командной строке. Если же нужно работать сразу с *несколькими* файлами — например для перемещения их в другой каталог с помощью `mv`, достраивание не помогает. Необходим способ задать одно «общее» имя, которое будет описывать сразу группу файлов, с которыми будет работать команда. В подавляющем большинстве случаев это можно сделать при помощи **шаблона**.

Шаблоны

Символы в шаблоне разделяются на обычные и **специальные**. Обычные символы означают сами себя, а специальные обрабатываются особым образом:

- Шаблону, стостоящему только из обычных символов, соответствует *единственная* строка, состоящая из тех же символов в том же порядке. Например, шаблону «`abc`» соответствует строка `abc`, но не `aBc` или `ABC`, потому что большие и маленькие буквы различаются.

- Шаблону, состоящему из единственного спецсимвола «`*`», соответствует *любая* строка любой длины (в том числе и пустая).

- Шаблону, состоящему из единственного спецсимвола «`?`», соответствует *любая* строка длиной в *один* символ, например, `a`, `+` или `@`, но не `ab` или `8888`.

- Шаблону, состоящему из любых символов, заключённых в квадратные скобки «`[`» и «`]`» соответствует строка длиной в *один* символ, причём этот символ должен *встречаться* среди заключённых в скобки. Например, шаблону «`[bar]`» соответствуют только строки `a`, `b` и `r`, но не `c`, `B`, `bar` или `ab`. Символы внутри скобок можно не перечислять полностью, а задавать *диапазон*, в начале которого стоит символ с наименьшим ASCII-кодом, затем следует «`-`», а затем — символ с наибольшим ASCII-кодом. Например, шаблону «`[0-9a-fA-F]`» соответствует одна шестнадцатеричная цифра (скажем, `5`, `e` или `C`). Если после «`[`» в шаблоне следует «`!`», то ему соответствует строка из одного символа *не* перечисленного между скобками.

- Шаблону, состоящему из *нескольких* частей, соответствует строка, которую можно разбить на столько же подстрок (возможно, пустых), причём первая подстрока будет отвечать первой части шаблона, вторая — второй части и т. д. Например, шаблону «`a*b ?c`» будут соответствовать строки `ab@c` («`*`» соответвтует пустая подстрока), `a+b=c` и `aaabbc`, но не соответствовать `abc` («`?`» соответвтует подстрока `c`, а для «`c`» соответствия не находится), `@ab@c` (нет соответствия для «`a`») или `aaabbbc` (из трёх `b` первое соответствует «`b`», второе — «`?`», а вот третье приходится на «`c`»).

Использование шаблонов

Шаблоны используются в нескольких конструкциях shell. Главное место их применения — командная строка. Если оболочка видит в командной строке шаблон, она немедленно заменяет его на список *файлов*, имена которых ему соответствуют. Команда, которая затем вызывается, получает в качестве параметров список файлов уже безо всяких шаблонов, как если бы этот список пользователь ввёл вручную. Эта способность командного интерпретатора называется **генерацией имён файлов**.

```
[methody@localhost methody]$ ls .bash*
 .bash_history  .bash_logout  .bash_profile  .bashrc
[methody@localhost methody]$ /bin/e*
 /bin/ed /bin/egrep /bin/ex
[methody@localhost methody]$ ls *a*
 sample-file
[methody@localhost methody]$ ls -dF *[ao]*
 Documents/  examples/  loop  to.sort*
```

При использовании шаблонов новичок может натолкнуться на несколько «подводных камней». В приведённом примере только первая команда срабатывает *не* вопреки ожиданиям: шаблон «`.bash*`» был превращён командной оболочкой в список файлов, начинающихся на `.bash`, этот список получила в качестве параметров командной строки утилита `ls`, после чего честно его вывела. «`/bin/e*`» — это на самом

деле опасная команда, с которой в данном случае просто повезло: этот шаблон превратился в список файлов из каталога `/bin`, начинающихся на «`e`», и *первым* файлом в списке оказалась безобидная утилита `/bin/echo`. Поскольку в командной строке ничего, кроме шаблона, не было, именно строка `/bin/echo` была воспринята оболочкой в качестве *команды*, которой — в качестве *параметров* — были переданы *остальные* элементы списка — `/bin/ed`, `/bin/egrep` и `/bin/ex`.

Что же касается `ls *a*`, то кажется, что эта команда должна была выдать список файлов в текущем каталоге, имя которых *содержит* «`a`». Вместо этого на экран вывелось имя файла из подкаталога `examples`... Впрочем, никакой чёрной магии тут нет. Во-первых, имена файлов вида «`.bash*`» хотя и содержат «`a`», но начинаются на точку, и, стало быть, считаются **скрытыми**. Скрытые файлы попадают в список сгенерированных имён только если точка в начале указана *явно* (как в первой команде примера). Поэтому по шаблону «`*a*`» в домашнем каталоге `bash` нашёл только подкаталог с именем `examples`, его-то он и передал в качестве параметра утилите `ls`. Что вывелось на экран в результате образовавшейся команды `ls examples`? Конечно, содержимое каталога. Шаблон в последней команде из примера, «`*[ao]*`», был превращён в список файлов, чьи имена содержат «`a`» или «`o`» — `Documents examples loop to.sort`, а ключ «`-d`» потребовал у `ls` показывать информацию о каталогах, а не об их содержимом. В соответствии с ключом «`-F`», `ls` расставил «`/`» после каталогов и «`*`» после исполняемых файлов.

Ещё одно отличие генерации имён от стандартной обработки шаблона — в том, что символ «`/`», разделяющий элементы пути, *никогда* не ставится в соответствие «`*`» или диапазону. Происходит это не потому, что искажён алгоритм, а потому, что при генерации имён шаблон применяется именно к элементу пути, внутри которого уже нет «`/`». Например, получить список файлов, которые находятся в каталогах `/usr/bin` и `/usr/sbin` и содержат подстроку «`ppp`» в имени, можно с помощью шаблона «`/usr/*bin/*ppp*`». Однако *одного* шаблона, который бы включал в этот список ещё и каталоги `/bin` и `/sbin` — то есть подкаталоги *другого* уровня вложенности — по стандартным правилам сделать нельзя[1].

[1] Генерация имён файлов в **zsh** предусматривает специальный шаблон «`**`», которому соответствуют подстроки с *любым* количеством «`/`». Пользоваться им следует *крайне осторожно*, понимая, что при генерации имён по такому шаблону выполняется операция, аналогичная не `ls`, а `ls -R` или `find`. Так, использование «`/**`» в начале шаблона вызовет просмотр *всей* файловой системы!

Если перед любым специальным символом стоит «`\`», этот символ лишается специального значения, **экранируется**: пара «`\символ`» заменяется командным интерпретатором на «`символ`» и передаётся в командную строку безо всякой дальнейшей обработки:

```
[methody@localhost methody]$ echo *o*
 Documents loop to.sort
[methody@localhost methody]$ echo \*o\*
 *o*
[methody@localhost methody]$ echo "*o*"
 *o*
[methody@localhost methody]$ echo *y*
 *y*
[methody@localhost methody]$ ls *y*
 ls: *y*: No such file or directory
```

Обратите внимание, что шаблон, которому не соответствует *ни одного* имени файла, `bash` раскрывать не стал, как если бы все «`*`» в нём были экранированы. В самом деле, какое из двух зол меньшее: изменять *интерпретацию* спецсимволов в зависимости от содержимого каталога, или сохранять логику интерпретации с риском превратить команду с параметрами в команду *без параметров*? Если бы, допустим, шаблон, которому не нашлось соответствия, попросту удалялся, то команда `ls *y*` превратилась бы в `ls` и неожиданно выдала бы содержимое всего каталога. Авторы `bash` (как и Стивен Борн, автор самой первой командной оболочки — `sh`) выбрали более непоследовательный, но и более безопасный первый способ[1].

Лишить специальные символы их специального значения можно и другим способом. Разделители (пробелы, символы табуляции и символы перевода строки) перестают восприниматься таковыми, если часть командной строки, их содержащую, окружить двойными или одинарными кавычками. В кавычках перестаёт «работать» и генерация имён (как это видно из примера), и интерпретация других специальных символов. Двойные кавычки, однако, допускают выполнение **подстановок** переменной окружения и результата работы команды.

[1] Авторы **zsh** пошли по другому пути: в этой версии shell использование шаблона, которому не соответствует ни одно имя файла, приводит к *ошибке*, и соответствующая команда не выполняется.

Глава 8

Графический интерфейс в Linux

Максим Отставнов, Кирилл Маслинский

Оконная система X и её реализации

Графический интерфейс не является неотъемлемой частью Linux — это просто одна из её компонент, такая же необязательная с точки зрения архитектуры системы, как, например, программа для рисования изображений. Но для тех программ, которые используют графические ресурсы, эта компонента предоставляет возможность работать с графическими объектами (линиями, прямоугольниками, цветами), ничего не зная о деталях работы конкретных устройств графического вывода (видеокарты и монитора). Это похоже на то, как ядро скрывает от программ детали работы с конкретным оборудованием, например, жёстким диском, предоставляя им работать с файлами. Поэтому комплекс программ, предоставляющий доступ к графическим ресурсам, называют **графической подсистемой**. В Linux функции графической подсистемы выполняет оконная система «Икс».

Графическая подсистема с точки зрения операционной системы представляет собой группу обычных процессов, управление которыми производится общесистемными средствами. Точно так же, общесистемными средствами производится и управление процессами, запускаемыми «из-под» этой графической среды. Графическая подсистема отнюдь не монополизирует использование компьютера; параллельно с её работой продолжает исполняться множество служебных системных процессов; с других терминалов (если система многотерминальная)

могут запускаться другие программы или даже другие графические подсистемы.

Оконная система Икс (от англ. X window system, далее — просто X) — один из самых больших и успешных проектов в истории компьютерной техники — восходит к 1984 г., когда разработчики двух систем компьютерной графики, претендующих на универсальность — проектов Athena (Массачусетский технологический институт) и W Windowing (Стэнфордский университет) — решили объединить свои усилия. Подробнее об истории этого проекта можно узнать, например, из статьи в Wikipedia[1].

Тогда перед ними стояла задача создать систему компьютерной графики, позволяющую совместно использовать самые разные компьютерные платформы. Решением стало создание специального протокола X, который позволял разделить программы-клиенты и сервер, предоставляющий графические ресурсы, отсюда и возможность исполнять программу-клиент на одном компьютере, сервер на другом, а данные между ними передавать по сети.

Проект этот был настолько наукоёмок и настолько полно охватывал тогдашнюю область задач, связанных с графикой, что серьёзных альтернатив ему так и не возникло. С тех пор X прошла через одиннадцать основных релизов (отсюда другое название — X11, представляющее собой название и текущую версию протокола) и множество версий. И возникновение, и вся история развития X тесно связаны с ОС UNIX, а теперь, естественно, и Linux. Тем не менее, реализации X доступны и для нескольких альтернативных архитектур ОС, включая Windows NT.

Существует несколько реализаций X, дальнейшее изложение будет ориентировано на две широко распространённые свободные реализации: XFree86 и XOrg. XFree86 изначально создавалась для семейства процессоров Intel 386, и вплоть до 2004 года была самой популярной реализацией X. XFree86 поддерживает беспрецедентно широкий спектр оборудования (оно и понятно, учитывая существующий «зоопарк» видеокарт и устройств ввода для платформы PC). Благодаря доступности исходных текстов и пользовательской аудитории в десятки миллионов человек, XFree86 весьма устойчива и хорошо оттестирована, по крайней мере, насколько это возможно для такого разнообразия поддерживаемого оборудования.

[1] http://en.wikipedia.org/wiki/X_Window_System#History

В последние годы параллельно с XFree86 развивается основанная на тех же исходных текстах X Window System графическая подсистема XOrg. До недавнего времени по спектру поддерживаемого оборудования, архитектур и функциональности XOrg мало чем отличалась от XFree86, и сейчас они тоже примерно эквивалентны с точки зрения пользователя. Однако направление развития этих двух проектов, состав их разработчиков и лицензионная политика несхожи. В ближайшем будущем вполне вероятно, что XOrg обгонит XFree86 и по возможностям, и по частоте использования.

Большинство из того, о чем будет говориться в последующих разделах, справедливо для любой реализации X на любом оборудовании и под любой ОС, список которых можно найти на `http://www.X.org`.

Цветной бутерброд

Большинство пользователей, установив систему, получают в своё распоряжение готовую графическую среду. Однако то, что сидящему за монитором представляется сплошной графической операционной средой, реализовано как многослойный бутерброд технологий. Попробуем разобраться в его устройстве «по слоям».

«Чистая» X

Непосредственно с оборудованием (видеосистемой, устройствами ввода и динамиком) работает **X-сервер**. Перечисленное оборудование в совокупности называется **X-терминалом** (аппаратным X-терминалом называется и специализированный компьютер, на котором исполняется исключительно X-сервер). X-сервер захватывает оборудование и предоставляет возможность выводить на эти устройства и получать от них ввод другим программам — X-клиентам — по особому протоколу, который так и называется, X-протокол.

Здесь сразу видно важное технологическое достоинство X: взаимодействие X-сервера с X-клиентами происходит по специфицированному протоколу, который может туннелироваться через TCP/IP и, соответственно, клиенты и сервер могут исполняться на разных узлах сети. Это означает, что одни и те же программы могут эксплуатироваться в разных топологиях, включая совокупность автономных рабочих станций (персональных компьютеров), совокупность рабочих

станций без данных или бездисковых рабочих станций (локальная сеть), многопользовательскую систему с X-терминалами (или какую-либо смешанную топологию).

Ещё одним ресурсом, который предоставляет X-сервер, являются шрифты. Оперировать шрифтами он может самостоятельно, либо с помощью другой программы, которая обеспечивает масштабирование шрифтов — **фонт-сервера** (от англ. font — шрифт).

На рис. 8.1 показана «чистая» оконная система X — когда запущен только X-сервер и никаких приложений — то, с чем большинство пользователей никогда не сталкивается. Запустить её обычно можно, подав команду: `X &`.

Рис. 8.1. Чистая X

Мы видим традиционный серый экран с не менее традиционным курсором в виде буквы «x». Используя мышь или другое координатное устройство, курсор можно перемещать по экрану. На нажатие кнопок мыши и клавиш никакой видимой реакции не следует. И невидимой тоже — сервер готов передавать эти сигналы своим клиентам, а клиенты пока не запущены. Хотя на самом деле некоторые комбинации клавиш X перехватывает и обрабатывает. Это Zap (*Ctrl–Alt–Backspace*) — завершение работы сервера (если эта возможность не запрещена при конфигурации), Zoom (*Ctrl–Alt–+* и *Ctrl–Alt–-*) — «горячее» переключение доступных видеорежимов. В некоторых ОС (Например, GNU/Linux) *Ctrl–Alt* в сочетании с функциональной клавишей освобождает оборудование и передаёт его на время соответствующей **виртуальной консоли** — запущенной в неграфическом ре-

жиме (обычно, vga-режиме) программе, позволяющей пользователю зарегистрироваться в системе и получить доступ к командной строке оболочки.

Рис. 8.2. xterm в среде X

Воспользуемся последней возможностью, перейдём на консоль и запустим первое клиентское приложение: программу xterm (рис. 8.2). На экране X появилось окно, а в окне можно видеть интерфейс клиентского приложения. В данном случае интерфейс текстовый, а приложение представляет собой эмулятор терминала, на котором запущена командная оболочка системы по умолчанию. С эмулятором можно делать все то же, что и с обычным терминалом: издавать команды, получать результат и запускать другие программы. Если программы текстовые (строчные или оконные), исполняться они будут в том же окне, а если графические (как и сам xterm) — в отдельных окнах.

Запустим программу xclock (рис. 8.3). При её запуске мы использовали несколько параметров, задающих геометрию (местоположение и размер) вновь порождаемого окна, цвет его фона и шрифта по умолчанию, толщину и цвет рамки. Эти (и некоторые другие) параметры типичны для программ, построенных на основе графической библиотеки X Toolkit. Все значения параметров, заданные при вызове программы, кроме геометрии окна (местоположения на экране и размера), могут быть перекрыты самим запускающимся приложением. Дело в том, что окно выделяется клиентскому приложению при запуске, и все доступные ему ресурсы этим окном и ограничены — это свойство

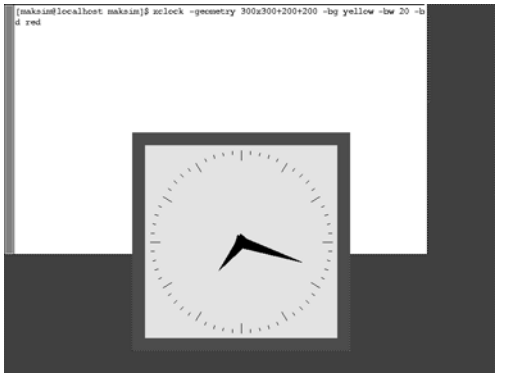

Рис. 8.3. xclock в среде X

X-протокола. Переместить окно или изменить его размер средствами «чистой» X невозможно.

Запустив несколько экземпляров того же xterm (и почитав документацию) можно обнаружить, что и «чистая» X умеет не так мало. Например, оперирует буфером обмена текстом между приложениями и предоставляет текстовым приложениям такой ресурс, как полосу прокрутки (полоса сбоку окна, при помощи которой можно листать текст вверх или вниз, щёлкая по ней левой и правой кнопками мыши соответственно, — это наследие проекта Athena).

Итак, ключевой компонент графической платформы — X-сервер:

- захватывает оборудование;
- создаёт по запросу других программ (которые в этой терминологии называются **X-клиентами**) окна;
- предоставляет другим программам возможность работы в окнах, т. е. вывода информации в эти окна и обработки сигналов от устройств ввода (клавиатуры и мыши или другого координатного устройства), когда окно, назначенное программе, является активным. Предоставление ресурсов возможно в том числе и через сеть, когда клиент и сервер работают на разных компьютерах (узлах).

В среде, образуемой X-сервером, окно, выделяемое клиенту, является фиксированным: его геометрия задаётся при запуске клиента и

сохраняется в течение всего сеанса работы с этим клиентом. Есть ли польза от системы, работающей с фиксированными окнами? Да, это вполне соответствует цели создания специализированных систем с графическим интерфейсом пользователя. Примером такой системы может служить терминал, позволяющий получать справки о расписании поездов и стоимости проезда, установленный на вокзале. Однако этих возможностей совершенно недостаточно для универсального «настольного» применения компьютера.

При универсальном применении компьютера характерна поочерёдная работа с различными программами (иногда достаточно большим их количеством), причём пользователь может отрываться, допустим, от редактирования текста, чтобы поработать с иллюстрацией при помощи другой программы, прочитать почту или заглянуть на интернет-страницу, затем возвращаться к редактированию текста и т. д. Эти возможности обеспечивает другая программа — **менеджер окон**, представляющая собой следующий «слой» в графической среде пользователя.

Менеджеры окон

Для одновременной и поочерёдной работы с разными программами, требуется возможность управлять окнами (с помощью клавиатуры или мыши), т. е. возможность изменять «на лету» их геометрию (положение и размеры), а также (обычно не относимое к геометрии) положение в воображаемой «стопке» окон — от этого зависит, какое из окон будет «верхним» (видимым полностью), если окна перекрывают друг друга на плоскости экрана.

Управление окнами и составляет основную функцию **оконного менеджера**. Устоявшийся англоязычный термин window manager, относящийся к этому классу программ, мы будем передавать далее словосочетанием-калькой «оконный менеджер», которое, впрочем, не представляется особенно удачным, так же, как и встречающиеся в литературе «менеджер окон», «администратор окон» и «диспетчер окон».

Технически ограничение на изменение геометрии однажды выделенного X-сервером окна преодолевается оконным менеджером за счёт того, что ему в качестве окна выделяется весь экран. Окно во весь экран может быть названо **корневым**, по аналогии с корневым каталогом в файловой системе. На самом деле, менеджер окон — не единственная программа, способная работать с корневым окном; например,

входящая в комплект поставки X утилита xsetroot позволяет установить цвет фона или поместить на него рисунок.

Прикладным программам, таким образом, выделяются далее уже не окна собственно X, а окна оконного менеджера. Изменение положения окна в подавляющем большинстве случаев ничего не требует от программы-клиента, однако желательно, чтобы программа была достаточно «сообразительной», чтобы изменить своё поведение при изменении размеров выделенного ей окна «на лету». Это справедливо для большинства, но не для всех программ (в частности, этого «не умеют» многие старые программы и некоторые компьютерные игры).

В свою очередь, и оконный менеджер может быть достаточно «умен», чтобы понять, что программа не реагирует на изменение геометрии окна, и запретить пользователю изменять размеры окна для данной программы (чтобы он не оказался в ситуации, когда ему видна лишь часть области вывода программы или наоборот, часть окна прикладной программы пуста). Однако такое решение может привести к весьма дискомфортным ситуациям (например, если при запуске программы её окно оказывается больше экрана)[1].

Менеджеров окон существует превеликое множество — под любой набор задач, которые может решать графическая многооконная система. Их настолько много, что выбрать какой-нибудь в качестве «типичного представителя семейства» затруднительно. Поэтому выберем один из самых развитых — Enlightenment[2].

«Просвещение» (англ. enlightenment) создано Карстеном Хайцлером и Джеффом Харрисоном (Carsten Haitzler, Geoff Harrison). До 2000 г. он был основным менеджером окон в популярной среде GNOME, затем уступив это место менее функциональной, но более быстрой «Рыбе-пиле» (Sawfish). Он продолжает оставаться GNOME-

[1] Следует отметить, что большинство базовых функций оконных менеджеров при исполнении опирается на поддержку оконной системой X функций двумерной графической акселерации (ускорения), реализованных практически во всех современных графических адаптерах. В отличие от трёхмерной акселерации, полезной лишь для достаточно узкого круга приложений (программ трёхмерного моделирования, компьютерных игр), двумерная акселерация — действительно универсальна и полезна для графического пользовательского интерфейса. При использовании карты без двумерного ускорения или карты, чья акселераторная функциональность не поддерживается системой X, можно рекомендовать настроить среду таким образом, чтобы исключить ресурсоёмкие функции, например, визуализацию перемещения окна со всем его содержимым, дабы избежать неоправданного роста нагрузки на процессор и драматического падения производительности.

[2] `http://www.enlightment.org`

Рис. 8.4. Оконный менеджер Enlightenment

Рис. 8.5. Оконное меню Enlightenment

совместимым, и многие пользователи этой популярной среды предпочитают его, хотя и без GNOME у Enlightenment поклонников хватает.

Запустим Enlightenment (рис. 8.4), набрав в командной строке xterm `enlightenment &`. Первое, что мы видим — это появившиеся вокруг окна нашего xterm элементы оформления: рамка и строка заголовка с кнопками. Окно теперь можно перемещать по экрану, «ухватив» за заголовок, масштабировать (изменять размер), «взяв» за бок или за угол, максимизировать, минимизировать или закрыть, нажав соответствующую кнопку. Спрашивается, что ещё можно делать с окном?

Вопрос не праздный. Нажав на левую кнопку в заголовке, получаем неожиданно разнообразное меню таких действий (рис. 8.5). Оказывается, его можно ещё уничтожить (Annihilate), поднять/опустить (Raise/Lower), скрутить/раскрутить (Shade/Unshade) приклеить/отклеить (Stick/Unstick) и выполнить ещё массу действий, для которых потребовались отдельные меню. Набор этих действий зависит от конкретного менеджера окон (и Enlightenment — один из самых богатых возможностями), а то, какие из них выведены в строку заголовка отдельными кнопками, — от того, как он настроен (пользователем или по умолчанию).

Базовая (а также расширенная) функциональность оконных менеджеров доступна пользователю прежде всего за счёт введения в интерфейс так называемых **виджетов** (от англ. widgets, сокращение от window gadgets, «оконные приспособления»). Виджеты — это рамки, кнопки, меню и пр., которые служат «органами управления» окна. Технически (в терминах оконной системы X) виджеты представляют собой отдельные окна, примыкающие к окну прикладной программы и, как правило, перемещающиеся вместе с ним (рис. 8.6).

Обрамление окна обычно составляют следующие элементы:

`Рамка, окружающая окно`

При «буксировке» рамки мышью окно изменяет свой размер. Иногда для изменения размера окна предназначены только выделенные «уголки» рамки, представляющие собой отдельные виджеты.

`Полоса заголовка`

Часто совпадает с одной из (обычно, верхней) сторон рамки. В полосе заголовка может содержаться название программы или запустившая программу команда, а также другая информация, специфичная для окна. При «буксировке» полосы заголовка переме-

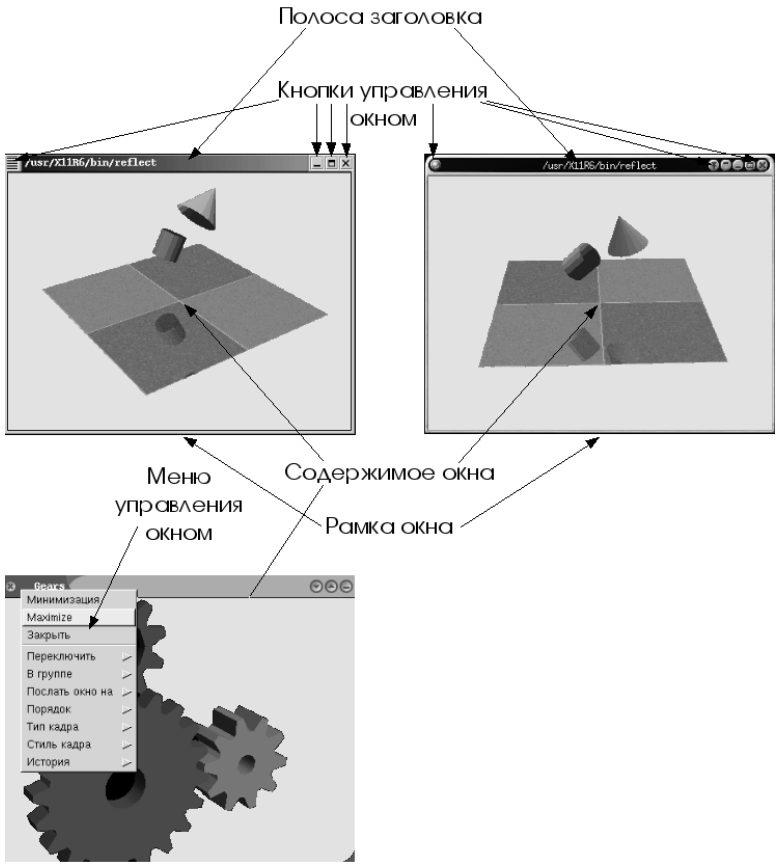

Рис. 8.6. Виджеты

щается все окно. Со «щелчками» различными кнопками мыши на полосе заголовка также могут быть связаны различные действия по управлению окнами.

Кнопки управления окном

Часто вынесенные на полосу заголовка или в другое место рамки кнопки позволяют выполнить с ним такие действия, как закрытие (часто сопровождающееся выходом из программы, открывшей

окно), максимизация (разворачивание окна на весь экран), минимизация/сворачивание, вызов меню управления окном, которое может содержать весьма обширный репертуар других действий.

Детали реализации обрамления окна могут быть весьма различными в зависимости от конкретного оконного менеджера и его настроек.

Откуда берутся такие ресурсы, как виджеты, их декор и способ поведения? Конечно, менеджер окон может содержать их в себе. Но такой подход не очень характерен для открытых систем, одним из принципов разработки которых является компонентность. Поэтому виджеты и их функции обычно объединяются в стандартные библиотеки (toolkits), которые могут быть использованы множеством различных программ. Большинство развитых менеджеров окон, менеджеров рабочего стола и разработанных специально для них приложений можно сгруппировать по библиотекам виджетов, с опорой на которые они разработаны.

С точки зрения пользователя виджеты, составляющие обрамление окна, часто воспринимаются как его часть. Однако не следует забывать, что внутри окна (содержимым которого управляет уже прикладная программа, а не менеджер окон) зачастую тоже есть свои виджеты: кнопки, полосы прокрутки, переключатели, меню и т. п. В общем случае, используемые оконным менеджером и прикладной программой библиотеки виджетов могут и не совпадать.

Собственно, управление окнами — основная функция оконного менеджера, и на этом его функциональность может и заканчиваться. Однако большинство из них выполняют по крайней мере ещё одну функцию.

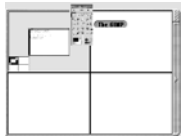

Рис. 8.7. Пейджер

Вы уже обратили внимание на то, что при запуске Enlightenment на экране появилось ещё одно окно. Это так называемый **пейджер** (pager), на рис. 8.7 он изображён крупным планом. На пейджере представлена миниатюрная копия экрана, обновляющаяся в режиме реального времени, причём, если подвести курсор к изображению отдельного окна, оно увеличивается и рядом высвечивается название приложения, запущенного в нем. Но почему экран занимает только четверть окна пейджера? Потому что оконный менеджер позволяет оперировать «виртуальным» столом (от англ. virtual desktop, также

рабочим столом), по размеру превышающим физический экран, а пейджер — одно из средств перемещения физического экрана по рабочему столу. Enlightenment позволяет создавать до 64 экранов на рабочем столе.

Менеджер окон, который помимо управления окнами обладает рядом дополнительных функций, может использоваться в качестве операционной графической среды пользователя, предоставляющей полный спектр возможностей для параллельной работы с несколькими задачами. Наиболее часто такими дополнительными функциями являются следующие:

`Минимизация/сворачивание окон и управление свёрнутыми окнами`

Работа на экране, «захламлённом» десятком различных окон, может быть дискомфортной, и крайне полезна возможность **свернуть** (минимизировать) окно со временно неиспользуемой программой. Для того, чтобы средствами графической среды можно было окно затем развернуть, оно и в свёрнутом состоянии должно каким-то образом отображаться. Существует несколько относительно распространённых способов отображения свёрнутых окон. Например, «на столе» может оставаться полоса заголовка свёрнутого окна, по щелчку на которой оно вновь разворачивается. Свёрнутым окнам могут соответствовать **пиктограммы** (иконки, значки) на поверхности рабочего стола или в специально отведённом для этого окне (**панели управления**). Свёрнутые окна могут отображаться как пункты общего или специального меню (см. ниже).

`Управление несколькими рабочими столами`

Практика показывает, что для многих продвинутых пользователей, которые осваивают открытые системы, уже имея опыт работы в характерных для ПК альтернативных системах, именно возможность работать на нескольких рабочих столах оказывается решающим плюсом оконной системы X. Действительно, переключение между виртуальными рабочими столами позволяет организовать комфортную работу со множеством программ даже на мониторах с относительно низким разрешением (1024x728, 800x600) и физическими размерами (17, 15-дюймовыми). В иных условиях комфортность работы существенно снизилась бы, или настоятельной необходимостью стало бы приобретение более крупного и ёмкого монитора (что зачастую влечёт за собой необходимость смены графической карты и прочих недешёвых мероприятий). Все современные оконные менеджеры поддерживают виртуальные рабочие столы, правда называются они везде по-разному: столы, рабочие области или экраны. До предела (чтобы не сказать, до абсурда) эта функциональность развита в оконном менеджере Enlightenment, который позволяет организовать до 64 экранов на рабочем столе, при этом рабочих столов также может быть более одного (точнее, до 32). Трудно представить, зачем может понадобиться две тысячи с лишним отдельных экранов (как правило, четырёх экранов хватает с избытком для любых практических задач), однако возможности приёма демонстрируются этим в полный рост.

`Быстрый запуск команд`

Возможность быстрого запуска предуготовленных команд обычно ассоциируется с **общим меню** (главным меню), вызываемым щелчком мыши на особом виджете, не связанном с прикладными окнами, или в свободной от прикладных окон области экрана.

`Настройка внешнего вида и поведения среды`

Поведение — реакция отдельных виджетов на операции с ними, модель фокусировки (способ переключения **активного** в данный момент окна, с которым связан ввод с клавиатуры и мыши) и т. п. Поведение и внешний вид оформления окон, а также наличие на экране общих виджетов, не связанных с конкретными прикладными окнами, **обои** (цвет фона или изображение в корневом окне) и т. п. могут варьировать в очень широких пределах. Иногда возможности такой настройки считают некими «архитектурными излишествами», однако более взвешенной является точка зрения, согласно которой в хорошем визуальном дизайне (так же, как и в хорошей архитектуре) ничто не является излишеством.

Пример возможностей настройки менеджера окон даёт тот же Enlightenment. Его меню настройки можно увидеть, щёлкнув правой кнопкой мыши на фоне экрана (рис. 8.8). Исследовав возможности настройки, можно обнаружить, что сказанное выше о способах работы с этим менеджером окон весьма условно, потому что поменять можно буквально все, от декора виджетов до количества и функций элементов оформления окон и их реакции на различные действия.

Лишь один пример: сколько способов визуализировать перемещение окна вы знаете? Разработчики Enlightenment придумали целых

Рис. 8.8. Меню настройки Enlightenment

шесть, включая фантастический «полупрозрачный». Настройки и расширения Enlightenment можно объединять в **темы** (англ. themes) и обмениваться ими.

Выше речь шла о таких свойствах Enlightenment, которые предположительно являются общими для всех оконных менеджеров. Но при этом чаще обычного употреблялись слова «обычно», «как правило», «может» и т. п. Это связано с чрезвычайным разнообразием решений на базе распространённых оконных менеджеров. Ниже более подробно и определённо речь пойдёт ещё о нескольких оконных менеджерах.

Оконные менеджеры BlackBox и FluxBox

BlackBox — один из самых компактных и быстродействующих оконных менеджеров. Он позволяет эффективно организовать работу на рабочем столе, не «захламляя» его ненужными ссылками и не расходуя экранное пространство на отображение громоздких элементов оформления (рис. 8.9).

Наряду с базовой функциональностью, BlackBox предоставляет (факультативно) панель, содержащую кнопки переключения между рабочими столами (по умолчанию их четыре) и заголовки открытых окон. Общее меню вызывается щелчком правой кнопкой мыши на свободном от окон месте рабочего стола. Меню (или любое из вложенных в него меню) щелчком по заголовку может быть превращено в окно,

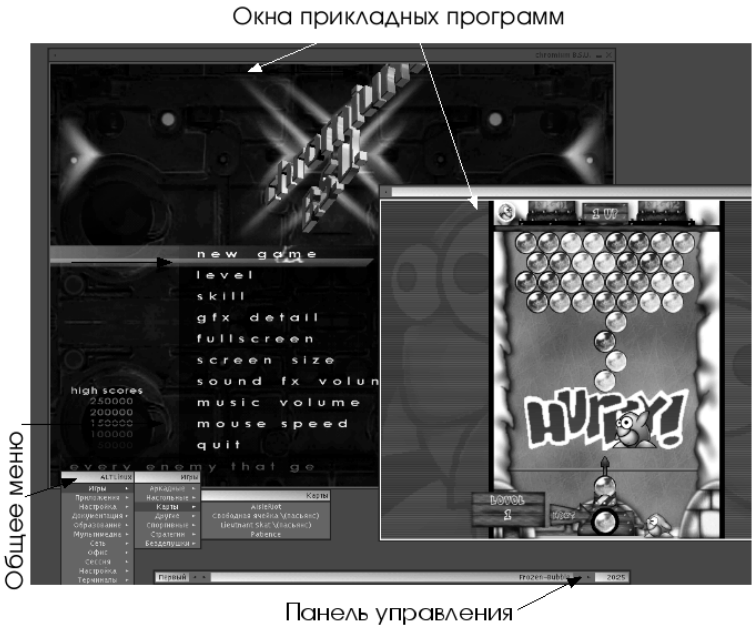

Рис. 8.9. Оконный менеджер BlackBox

остающееся на экране до явного его закрытия щелчком на соответствующей кнопке.

По умолчанию на полосе заголовка каждого окна присутствуют кнопки сворачивания (сворачивание можно выполнить также двойным щелчком на самом заголовке), максимизации и закрытия окна. Свёрнутое окно присутствует на экране в виде полосы заголовка, развернуть его можно повторным двойным щелчком на полосе заголовка или из меню «Workspaces» (рабочие области), доступного по щелчку средней кнопкой мыши на свободном от окон месте рабочего стола. Это же меню позволяет перейти на другой стол, добавить или удалить стол из рабочего пространства.

BlackBox поддерживает различные модели фокусировки ввода. Фокусировка по щелчку мыши (от англ. click to focus) позволяет реализовать стиль работы, привычный для пользователей KDE: окно становится активным (принимающим текущий ввод с клавиатуры и от мы-

ши) после щелчка на нем. Активное окно автоматически становится верхним (видимым полностью, даже если оно частично перекрывается с другими окнами). Небрежная фокусировка (от англ. sloppy focus) предполагает активизацию окна при попадании на него курсора мыши (окно при этом не «всплывает» автоматически наверх).

Наряду с панелью и конвертируемыми в дополнительные окна-панели меню, BlackBox реализует ещё один автономный виджет — так называемую **щель** (англ. slit). Щель располагается на краю видимого экрана и может содержать маленькие (без обрамления) окна специализированных программ (их существует около десяти), индицирующих какие-либо состояния среды или позволяющих быстро выполнить часто исполняемые действия.

На основе BlackBox созданы два более развитых оконных менеджера — OpenBox и более популярный FluxBox.

Внешний вид BlackBox, FluxBox и OpenBox легко настраивается с помощью механизма тем рабочих столов.

Оконный менеджер WindowMaker

WindowMaker (WM) — это свободная реализация (в рамках проекта GNUStep) концепций NextSTEP — первой получившей более или менее широкую известность универсальной графической среды пользователя. За недоступностью оригинальной NextSTEP для современных платформ, познакомиться с WM полезно и поучительно вне зависимости от того, собираетесь ли вы с ним работать. Это позволит увидеть исходную точку развития графических сред и оценить продуктивность (или непродуктивность) того, с чем эти идеи стали ассоциироваться со временем.

Основным автономным виджетом WM, как и NextSTEP, является **пирс** прикладных программ, представленный при запуске пиктограммой со скрепкой. При запуске любой корректной (с точки зрения WM), а также некоторых некорректных программ, кроме её окна на экране появляется её пиктограмма. Если «пришвартовать» эту пиктограмму к пирсу, она там и останется, позволяя запускать эту программу вновь и вновь простым щелчком по ней — это разработанный для NextSTEP интегрирующий интерфейс.

WM позволяет работать с несколькими рабочими столами (переключение по умолчанию при помощи клавиш *Alt–n* или через меню, доступное по щелчку правой кнопкой мыши на свободном месте

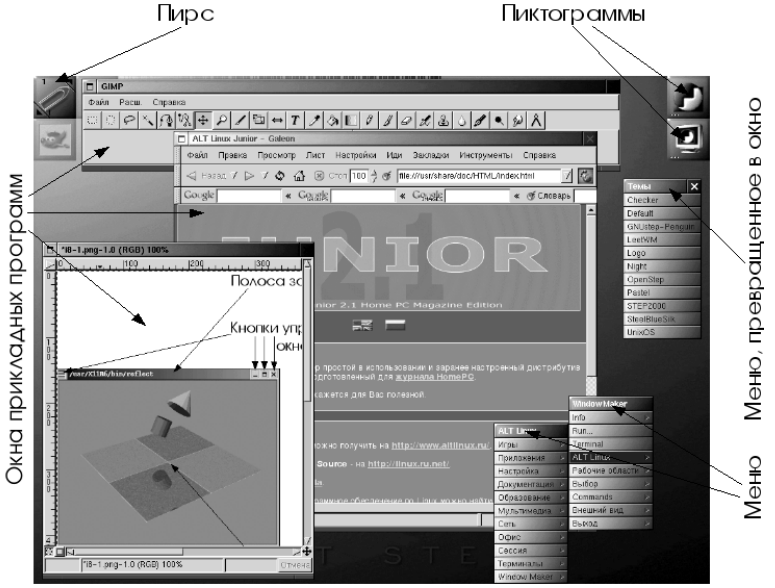

Рис. 8.10. Оконный менеджер WindowMaker

рабочего стола). WM очень гибко настраивается, как в отношении внешнего вида, так и в отношении поведения, причём большая часть настроек доступна из специальной графической утилиты, запускаемой по щелчку на пиктограмме с изображением ступеньки (рис. 8.10).

Оконный менеджер IceWM

IceWM — простой оконный менеджер, его очень часто выбирают пользователи, переходящие с Microsoft Windows или OS/2, поскольку он достаточно точно повторяет основные черты привычной для них графической рабочей среды (рис. 8.11).

Из автономных виджетов прежде всего стоит отметить панель с кнопкой, вызывающей главное меню (подобно тому, как это делает кнопка в Microsoft Windows, GNOME или KDE). С помощью панели можно также управлять текущим сеансом и настраивать IceWM. Впрочем, основное меню также доступно и по щелчку правой кнопкой

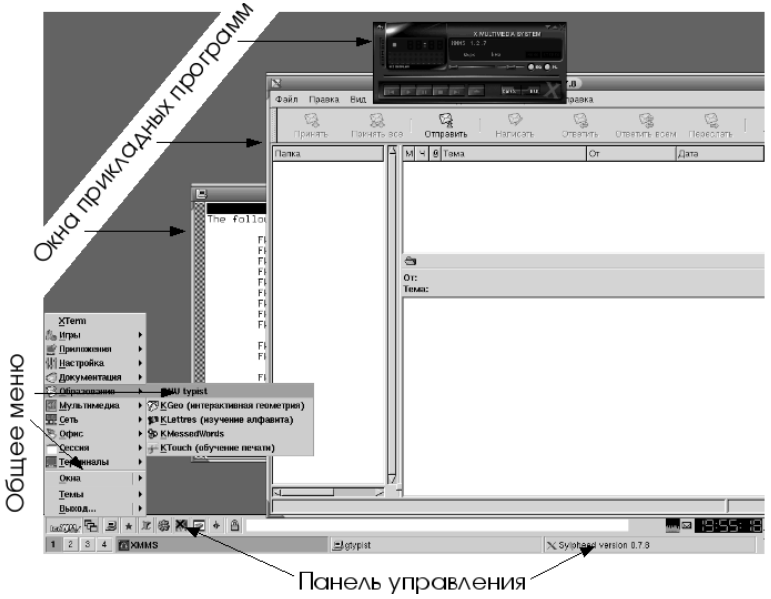

Рис. 8.11. Оконный менеджер IceWM

мыши на свободном месте рабочего стола, что более привычно для пользователей WindowMaker, Sawfish, BlackBox или Enlightenment.

Панель также содержит список запущенных программ (включая те, окна которых минимизированы), на неё можно вывести и **мини-терминал**, позволяющий оперировать командной строкой. Для выполнения любого действия может быть назначена специальная клавиатурная комбинация.

IceWM также позволяет работать с множеством рабочих столов (рабочих мест), которые нумеруются или именуются пользователем.

Интегрированные графические среды

Существует два подхода к тому, как можно достроить оконную систему до полнофункциональной среды, позволяющей пользователю решать все (или почти все) его практические задачи. Во-первых, можно расширить функциональность менеджера окон, добавив в него

недостающие возможности. Чего не хватает в оконном менеджере до полнофункциональной среды? Возможности запускать программы и утилиты. Достигается это обычно при помощи организации специального меню. С примером этого подхода мы уже познакомились в предыдущем разделе: этим путём пошли разработчики Enlightenment и ряд других проектов. Во-вторых, можно добавить в «графический бутерброд» ещё один слой — **менеджер рабочего стола** — работающий «поверх» менеджера окон и использующий функциональность последнего. Этим путам идут команды разработчиков GNOME и KDE, и для пользователей этих графических сред «бутерброд» становится уже трёхслойным: оконная среда X, оконный менеджер и менеджер рабочего стола.

С точки зрения пользователя нет чёткой границы между менеджерами окон с расширенной функциональностью и менеджерами рабочего стола, работающими «поверх» менеджера окон, поскольку они обеспечивают одну и ту же функциональность и нередко даже графически организованы сходным образом. Оба варианта предоставляют пользователю возможность работать в **графической среде** (desktop environment).

Интегрированная графическая среда предполагает не только единство оформления, но и трактовку объектов в рабочем пространстве (окон, файлов, пунктов меню и т. п.) как физических объектов, которые можно перемещать, выбрасывать в «корзину» и т. д. Однако сколько-нибудь последовательной теории интегрированных графических сред не существует. Изучая отдельные среды в динамике их развития, можно, тем не менее, выделить несколько общих черт.

- Они опираются на определённый интерфейс разработчика (API), состоящий из библиотек, доступных также разработчикам прикладных программ (будь то MS Windows API для Microsoft Windows, Motif для CDE, Qt для KDE или GTK+ для GNOME);

- Они реализуют элементы объектной метафоры: файлы, процессы (их потоки ввода-вывода) изображаются как отдельные объекты, на них можно фокусироваться и выполнять с ними различные действия, их состояния и изменения этих состояний также могут визуализироваться или озвучиваться. Целостная объектная метафора своей реализации не нашла (и, видимо, последовательно объектная среда была бы крайне неудобной в использовании).

- Они реализуют единообразные элементы управления (виджеты), зачастую не только в оформлении отдельных окон, но и в их содержимом.
- Они содержат те или иные элементы управления, не привязанные к отдельным окнам прикладных программ (общие меню, панели управления, поверхность стола и т. п.).
- Они позволяют согласованно изменять свойства интерфейса образующих среду программ (менеджера окон, менеджера рабочего стола, приложений, разработанных специально для данной среды).
- Они реализуют **буфер обмена**, позволяющий передавать типизованные данные от программы программе (оконная система X содержит буфер, позволяющий передавать данные лишь простого текстового типа).
- Они реализуют возможность «перетаскивания» при помощи мыши (drag'n'drop) объектов или данных между окнами одной программы или разных программ.

Однородность опыта при работе в интегрированных средах и связанная с нею привычность (иногда ошибочно называемая «интуитивностью», хотя она не имеет отношения к философскому и психологическому понятиям интуиции) позволяют при освоении нового инструмента-программы сосредоточиться на её прикладной логике, не задумываясь и специально не фокусируя внимания на приёмах работы, общих для всех инструментов. Это позволяет новому пользователю гораздо быстрее осваивать прикладные программы (делает более «крутой» пресловутую кривую обучения)[1].

Как ни парадоксально, основной недостаток работы в интегрированной среде является оборотной стороной основного достоинства: жёстко закреплённые навыки мешают при выходе за её пределы. Конечному пользователю, ограниченному опытом работы в одной среде, недостаёт «стереоскопичности» видения, глубины понимания; элементы эргономической логики могут напрямую ассоциироваться с определёнными визуальными элементами и «жестами», с помощью которых подаются команды.

[1] Разумеется, это сильно идеализированная картина. Иногда прикладная логика диктует некоторые элементы эргономики. Например, интерфейсы большинства систем автоматизированного конструирования и проектирования (CAD, САПР) весьма сходны, вне зависимости от среды, в которой работают эти программы.

Общеизвестны сложности, с которыми сталкиваются люди, долгое время работавшие в одной графической среде, при необходимости поработать в другой (пусть даже и весьма схожей). Для преодоления таких сложностей крайне полезным представляется знакомство с *разными средами* уже на начальном этапе освоения графических интерфейсов. Это не обязательно должны быть разные интегрированные среды, но само представление о том, что один и тот же результат может достигаться с помощью разных интерфейсных средств весьма важно. В общем случае это возможно и в рамках одной интегрированной среды из числа рассматриваемых ниже — и KDE, и GNOME в высшей степени гибки в отношении настройки внешнего вида и поведения.

На сегодня существуют и развиваются две свободные интегрированные графические среды общего назначения: KDE и GNOME. Они входят в поставку большинства стандартных (открытых) ОС, как свободных, так и несвободных. Хотя GNOME на полгода моложе KDE, мы начнём обсуждение именно с GNOME.

GNOME

GNOME (GNOME, GNU Network Object Model Environment — «Среда ГНУ, основанная на модели сетевых объектов», но также и «Образцовая среда для сетевых объектов ГНУ») — один из самых амбициозных и масштабных проектов в программистском сообществе.

Кроме реализации функционально полной графической среды (возможно, уместнее говорить о сенсуальных средах, учитывая то, что звук стал их полноправной частью), GNOME претендует на то, чтобы полностью реализовать спецификации промышленной платформы сетевого взаимодействия CORBA и полностью абстрагировать слой менеджера рабочего стола (или графической среды) от низлежащего слоя управления окнами (менеджера окон).

GNOME поддерживает ряд оконных менеджеров, среди которых: Sawfish (оконный менеджер по умолчанию), Enlightenment, IceWM, WindowMaker, AfterStep и FVWM2, совместимые с GNOME, впрочем, в разной степени.

Сегодняшняя версия GNOME — полноценная интегрированная среда, включающая реализацию повседневно необходимых функций и

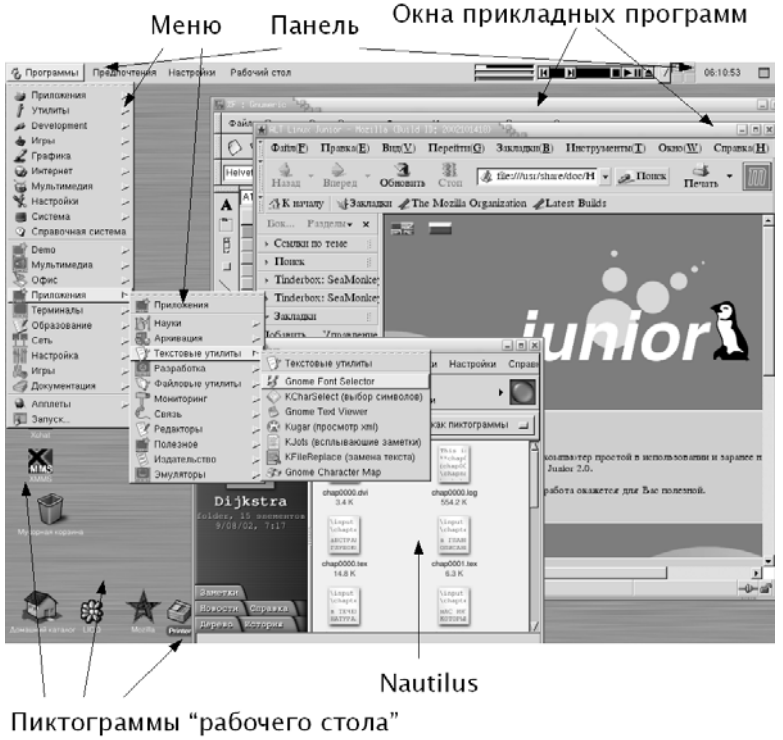

Рис. 8.12. Интегрированная среда GNOME

позволяющая использовать сторонние решения для реализации функциональности, которая в ней отсутствует (рис. 8.12).

GNOME использует один из самых развитых интерфейсных пакетов GTK+, реализованный для разных платформ. Над ним надстраивается масса компонентов и библиотек, обеспечивающих сетевую функциональность, интерфейсы к различным языкам программирования, работу со звуком через механизмы ОС и пр. Сам GNOME стремится оставаться мобильным и доступным во всех открытых системах. Он стабильно работает в GNU/Linux, BSD, AIX и Solaris; последнее обстоятельство способствовало поддержке разработки GNOME, которую оказывает Sun Microsystems через созданный в 2001 году «Фонд

GNOME», среди учредителей которого также крупнейшие поставщики свободных ОС.

С пользовательской точки зрения GNOME предстаёт как набор базовых компонентов интерфейса и **аплетов**, утилит и прикладных программ. К базовым компонентам относятся менеджер файлов и поверхности стола Наутилус (Nautilus), панели управления и меню GNOME Panel и центр управления (Gnome Control Center).

Менеджер файлов Nautilus позволяет отображать содержимое файлов и каталогов в окнах и выполнять над файлами обычные действия (удаление, переименование, копирование и перемещение и т. п.), а также осуществлять предварительный просмотр многих типов данных. Nautilus эффектен, но работа с ним не более эффективна, чем с прочими броузерами файлов, включаемыми обычно в графические среды (менеджер файлов CDE или Windows Explorer).

Помимо отображения содержимого каталогов в окнах, Nautilus также может отображать один из каталогов на поверхности рабочего стола: размещённые на нем иконки как бы приклеены к монитору, и при смене текущего экрана остаются на том же месте относительно наблюдателя (так же, кстати, ведут себя и открытые окна, если их «приклеить»).

Поддерживается широкий спектр операций переноса мышью (drag’n’drop), причём «перетаскивать» можно не только объекты (файлы, пункты меню и т. п.), но и некоторые их свойства: так, можно «взять цвет» в окне выбора цвета и перенести его на панель, которая воспримет его. Есть даже операции, позволяющие назначить один объект свойством другого: например, если на панель «перетащить» не цвет, а файл с картинкой, последняя станет её фоном. Переносить файлы между окнами Nautilus, на рабочий стол и панели можно практически без ограничений.

Уже упомянутые панели являются, наряду с менеджером файлов, важнейшей составной частью интерфейса GNOME. Панелей может быть неограниченное количество. Панель может быть двух типов: панель-меню (menu panel) и объектная панель (object panel). Первая из них содержит пункты меню и может содержать пиктограммы, а вторая — только пиктограммы. Последняя может быть краевой (edge), выровненной (aligned), скользящей (sliding) или плавающей (floating), но это скорее свойство панели (которое можно менять «на ходу»), определяющее особенности её поведения, чем тип.

Внешний вид и поведение панелей является в высшей степени конфигурируемым. Пользователь может задавать как глобальные предпо-

чтения (анимация движения панелей, отображение панельных объектов и пр.), так и индивидуальные предпочтения для каждой из них (её тип и положение на экране, ширина, возможность автосокрытия и принудительной минимизации, цвет и фоновое изображение и т. п.). Разумеется, пользователь может наполнять панели теми объектами, которые ему нужны.

На панелях могут присутствовать объекты пяти типов:

- **Аплет** (applet, «приложеньице») — интересный тип панельного объекта, демонстрирующий то, что он не обязан быть представлен статической картинкой. Это программа, места в панели которой достаточно, чтобы отображать какую-нибудь полезную (или забавную) информацию или даже принимать клавиатурный и/или координатный ввод. С GNOME поставляется масса аплетов, отображающих всякую полезную информацию (состояние ресурсов и статус сети, например) или позволяющих осуществить нетривиальные действия (например, mini commander, позволяющий набрать команду, не запуская терминала). Важными аплетами являются путеводитель по столу (Desktop Guide) и список задач (Task List), позволяющие переключаться между виртуальными экранами и активизировать окна запущенных программ соответственно.

- **Пускатель** (launcher) ассоциирован с приложением или командой, которые исполняются по щелчку на его пиктограмме в панели.

- **Выдвижной ящик** (drawer) — это кнопка, открывающая другую панель, перпендикулярно первой — некий аналог подменю в меню, который можно наполнить всевозможными аплетами.

- **Специальные объекты** — это те же аплеты, но выполняющие функции, которые другими средствами «достать» почему-либо нельзя (запереть экран, выйти из GNOME или запустить программу «вручную»). В качестве специальных объектов исполняются и программы, которые не были написаны специально для GNOME, но могут, тем не менее, осуществлять вывод в панель — **поглощённые программы** (swallowed applications).

- Наконец, **объект-меню** раскрывает меню.

За работу системы меню, как и за работу панелей, отвечает компонент GNOME Panel, и это не случайно: разница между панелью и меню в большей степени декоративная, чем сущностная. Любое меню можно зафиксировать на экране, и оно превратится в подобие панели-меню (только вертикальное, а не горизонтальное, и с меньшими возможностями настройки).

У GNOME нет единой иерархии меню: кроме главного, вызываемого объектом-меню с гномьей лапой (оно же, когда вызывается щелчком правой кнопки на фоне или нажатием клавиши, почему-то называется **глобальным** (global)), пользователь может создавать **обычные** (normal) меню, связанные с объектами-меню на панелях.

Меню настраиваются примерно так же, как и панели: пользователь может добавлять, менять и удалять пункты, создавать подменю и т. п. При этом создаваемые обычные меню изначально пусты, а главное/глобальное заполняется при установке всем, что GNOME найдёт в системе, и пользователю остаётся только убрать лишнее и переставить пункты в соответствии со своими предпочтениями.

Для настройки различных аспектов функционирования системы предназначен Центр управления, представляющий собой набор **управляющих аплетов** (каплетов, от англ. capplets, control applets), связанных с разными компонентами и прикладными программами.

Одни из них позволяют менять параметры рабочего стола и облик приложений (включая использование тем), другие — настраивать мультимедиа, третьи — управлять свойствами клавиатуры и мыши, и т. д.

Важным «каплетом» является менеджер т. н. **обработчиков документов** (Document Handlers), устанавливающий соответствие между типом файла или протокола и программой, выполняющей различные операции с ними. Набор каплетов является расширяемым, их можно разрабатывать не только для программ, написанных специально для GNOME, но и для внешних программ.

Также постоянно расширяется набор утилит, прикладных программ и аплетов, поставляемых с GNOME — вместе с программами, входящими в большинство дистрибутивов ОС, о которых GNOME «в курсе», их число превышает сотню. Перечислить их здесь нет никакой возможности, но среди них есть интерфейсы для администрирования системы, средства звукозаписи и воспроизведения, сетевые утилиты, игры и многое другое.

GNOME снабжён встроенной системой помощи; кроме того, его разработчиками совместно с Sun Microsystems подготовлено компакт-

ное руководство, доступное в разных форматах на сайте проекта. В его поставку входит система разработки графических приложений под GTK+, которая называется Glade и включает в себя специфические для GNOME элементы.

GNOME и большинство его компонентов соответствуют соглашениям об интернационализации и, соответственно, поддерживают работу с кириллицей, локализацию и перевод интерфейса.

KDE

Само название KDE (KDE, K Desktop Environment — «Графическая среда K») — явная пародия на CDE (Common Desktop Environment — «Общая настольная среда»). CDE была последней попыткой отрасли стандартизовать графическую среду на несвободной основе, предпринятой в конце девяностых годов. Буква «K» в KDE ничего не означает.

Несмотря на явно игривый тон, начинающийся с названия среды и продолжающийся в названии компонентов, KDE — очень серьёзный проект. В KDE любят играть со словами; например, универсальный броузер, входящий в среду, называется Konqueror (от англ. conqueror — «завоеватель», «покоритель»), терминал — Konsole (от console — «консоль»), а система помощи — вообще Kandalf (от имени Гэндальфа, мага из фантазийных произведений Дж. Р. Р. Толкиена).

Если единообразие и однородность графической среды считать достоинством, то KDE — несомненный лидер среди всех (как свободных, так и несвободных) интегрированных графических сред. Основное видимое средство интеграции — это универсальный броузер Konqueror. Функция Konqueror близка к той, которую приобрёл Windows Explorer — он совмещает функции гипермедийного броузера WWW и броузера локальных ресурсов (рис. 8.13).

Разработчики KDE пошли даже дальше своих коллег из Microsoft и определили ряд дополнительных протоколов, что позволило, в частности, просматривать с помощью броузера в единообразном формате все разнообразие справочной информации, представленное в сегодняшних открытых системах (традиционные страницы руководства `man`, гипертекстовую систему `Info` из проекта GNU, разрозненные файлы документации в текстовом и гипертекстовом формате). В Konqueror интегрирована также возможность предварительного просмотра содержимого большого количества типов файлов.

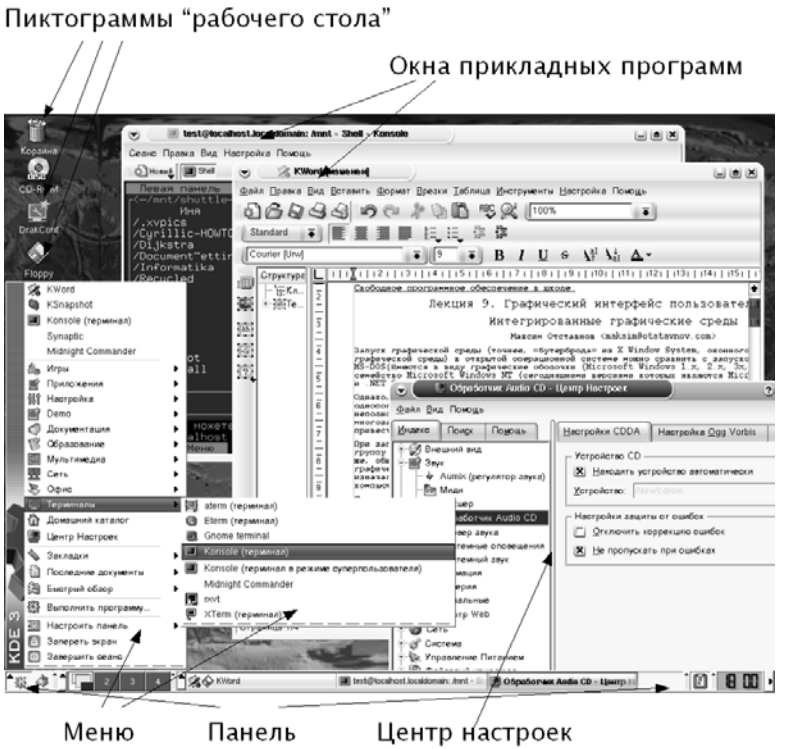

Рис. 8.13. Интегрированная графическая среда KDE

KDE включает также настраиваемую систему панелей и меню и интегрированный **центр управления**, позволяющий согласованно изменять параметры среды. KDE менее гибка в настройке, чем GNOME, однако её гибкости вполне достаточно для решения любых практических задач (в том числе, имитации вида и поведения других сред). KDE работает только с собственным оконным менеджером KWin.

В поставку KDE входит множество «аксессуаров» и прикладных программ, к тому же рядом с проектом выросла целая группа сопутствующих, ориентированных на те или иные предметные приложения, из которых самым развитым является офисный пакет KOffice.

Зачем нужны «лёгкие» среды?

В то время, как сама оконная система X много лет является фактическим отраслевым стандартом, лежащие «над» нею слои графической среды не стандартизованы. Какую-либо классификацию графических сред дать затруднительно, однако самым грубым образом их можно разделить на интегрированные и лёгкие. В последние годы разработчики GNOME и KDE работают над формулировкой общего открытого стандарта[1] на интегрированную графическую среду.

Оборотной стороной интегрированности является достаточно высокая их требовательность к ресурсам. Комфортная работа с KDE или GNOME последних версий начинается при производительности компьютера, примерно эквивалентной производительности 800 МГц процессора Celeron. Отказ от некоторых ресурсоёмких свойств (анимация изменений в среде и т. п.) позволяет снизить требования примерно до 500 МГц при объёме оперативной памяти от 128 МБ. Разумеется, эти цифры даже ниже характерных для компьютеров стартового уровня, поставляемых сегодня производителями, однако парк машин, находящихся в эксплуатации, как в офисах, так дома и в школе, включает и компьютеры с более низкими характеристиками.

Здесь помогут **лёгкие** графические среды, представляющие собой оконные менеджеры с расширенными возможностями. Описанные выше IceWM, BlackBox и FluxBox (а также чуть более требовательный к ресурсам WindowMaker)[2] позволяют достаточно комфортно работать с графикой на машинах производительностью (в эквиваленте Intel Pentium) примерно от 100 МГц и с памятью от 32 Мб.

Следует оговориться, что отказ от интегрированных графических сред не является панацеей: конкретные прикладные программы могут быть сами по себе достаточно требовательными к ресурсам. Кроме того, если прикладная программа изначально создана с ориентацией на определённую интегрированную среду, она может интенсивно использовать соответствующие библиотеки, даже если запускается в лёгкой среде. Например, запуск программ из пакета KOffice в лёгкой среде, на самом деле, даёт небольшой выигрыш по сравнению с его запуском из «родной» для него среды KDE.

[1] `http://freedesktop.org`

[2] Возможностью представить их обзор в компактном виде автор обязан прежде всего своим соавторам Егору Гребневу, Сергею Иванову, Михаилу Шигорину. — Прим. М. Отставнова.

Если необходимо задействовать имеющийся парк «слабой» техники для таких задач, а также, если необходимо сохранять в эксплуатации ещё менее производительные машины (например, старшие модели IBM PC-совместимых компьютеров на базе процессоров Intel 486 или AMD 586 или Макинтош на процессорах Motorola 68K), следует подумать об использовании такой техники в режиме графических терминалов или, по крайней мере, варианте запуска наиболее требовательных к ресурсам прикладных программ на сервере.

Следует оговорить также, что ограниченность аппаратных ресурсов не является единственным мотивом применения лёгких графических сред. Каждая графическая среда, интегрированная или лёгкая, обладает собственными уникальными особенностями, собственным стилем. Уместность использования каждой конкретной среды в значительной степени зависит от набора задач, решаемых на компьютере конкретным пользователем, и от его личных предпочтений.

См. также
Что такое открытые системы . . . . . [стр. 59]
X-терминалы: дома, в школе, в офисе . . . [стр. 195]
Прикладные графические программы [снаружи, стр. 50]

Глава 9

Своими руками

История одного скрипта

Антон Бояршинов

При использовании компьютера рано или поздно появляются задачи, которые могут быть автоматизированы. Часто для их решения можно найти специально разработанные программы. Но не для каждой задачи такую программу можно подобрать, так как задачи и представление о правильных способах их решения у всех разные. Часто эти задачи могут быть решены с помощью скриптов — небольших программ на языке командной оболочки или иных интерпретируемых языках.

Важным свойством таких программ является, на мой взгляд, возможность быстрого получения первых полезных результатов. Нередко 2–3-строчный скрипт способен сэкономить десятки минут монотонной ручной деятельности.

Ниже я привожу пример такого скрипта, сильно изменившего свою функциональность по мере развития и даже сменившего язык. Он, мягко говоря, не безупречен, так как я отнюдь не специалист по программированию на языке командной оболочки, в нём встречаются весьма странные и, возможно, не всегда безопасные решения. Но это реальный и живой пример того, что может сделать для себя в той или иной области почти каждый.

Я увлекаюсь фотографией. Последнее время — цифровой. Довольно быстро я столкнулся с тем, что просто сложить всё отснятое в один каталог — не самый оптимальный вариант размещения материала, так как по мере роста количества снимков найти нужный становится непросто. А хотелось бы также показывать, например, тематические подборки снимков гостям, легко записывать для них тематические компакт-диски.

Сначала я поискал среди готовых программ поддержки фотографических коллекций.

Первое решение было очень простым: надо разложить снимки по каталогам в соответствии с датой съёмки. Поскольку часть снимков подвергалась редактированию, дата модификации файла в качестве даты не годилась, и я использовал дату из EXIF (во многих графических форматах существует возможность записывать те или иные сведения об изображении, так большинство цифровых фотокамер сохраняет данные о параметрах съёмки, включая и дату).

На тот момент (2004 год) в результатах поиска `apt-cache search exif` нашлась лишь одна очевидная программа, которая могла мне помочь, но тогда мне и не требовалось большего. Вот первый вариант скрипта:

```
#!/bin/bash
for n in *.jpg;
  do
    date=‘exif $n | grep ’Date and Time  ’ \
    | cut -d’|’ -f2| cut -d’ ’ -f1| sed ’s/:/-/g’‘
    if [ -L ../days/$date/$n ]; then true;
    else
        mkdir -p ../days/$date
        ln -s ‘pwd‘/$n ../days/$date/$n
    fi
done
```

Первая строка указывает операционной системе на тот интерпретатор, которому следует передать для выполнения данную программу. Далее всё достаточно прозрачно, за исключением, пожалуй, волшебной строки: `date=‘exif $n | grep ’Date and Time ’| cut -d’|’ -f2| cut -d’ ’ -f1|sed ’s/:/-/g’‘`. В этой строке производится извлечение из файла и преобразование даты с последующим помещением её в переменную оболочки `$date`.

Получена она была примерно следующим путём (для понимания смысла используемых команд и их аргументов рекомендую заглянуть в соответствующие man-страницы):

```
[avb@boyarsh-book Animals]$ exif dsc_7622.jpg | grep ’Date and
Time  ’
Date and Time       |2006:01:21 22:06:34
[avb@boyarsh-book Animals]$ exif dsc_7622.jpg | grep ’Date and
Time  ’| cut -d’|’ -f2
2006:01:21 22:06:34
```

```
[avb@boyarsh-book Animals]$ exif dsc_7622.jpg | grep 'Date and
Time  '| cut -d'|' -f2|cut -d' ' -f1
2006:01:21
[avb@boyarsh-book Animals]$ exif dsc_7622.jpg | grep 'Date and
Time  '| cut -d'|' -f2|cut -d' ' -f1| sed 's/:/-/g'
2006-01-21
```

Однако через некоторое время этот скрипт перестал меня устраивать: я стал фотографировать в формат RAW, преобразовывая результат в JPEG при помощи программы ufraw. Доступная на тот момент её версия (а также версия, входящая в Compact 3.0) не поддерживала перенос данных EXIF из RAW-файла в получаемый из него JPEG. Пришлось искать иной способ переноса. Несмотря на то, что RAW-файлы от моей фотокамеры являются по сути TIFF-файлами специального вида, и дополнительная информация в них хранится стандартным образом и доступна для ряда программ, работающих с TIFF, применить их для переноса информации не удалось. Поиск в Сети тоже не сразу дал результат, так как мне не удалось придумать хороший запрос. Однако через некоторое время подходящий инструмент был найден. Он содержится в пакете perl-Image-ExifTool.

После нескольких проб и ошибок скрипт приобрёл примерно такой вид, как показано в примере 9.1 (на следующей странице).

Идея добавленного фрагмента состоит в том, что если в данных EXIF не содержится дата, и в текущем каталоге присутствует файл с таким же именем, но имеющий расширение. nef (RAW-файлы от камер Nikon имеют именно такое расширение), производится копирование метаданных из RAW-файла в соответствующий JPEG, после чего RAW-файл, если возможно, переносится в другой каталог. Для того чтоб перенести метаданные EXIF с RAW-файлов, выгруженных на оптические носители либо находящихся в другом каталоге, я реализовал возможность указать каталог, содержащий RAW-файлы, при помощи переменной командной оболочки.

Вскоре я понял, что простое раскладывание фотографий по каталогам в соответствии с датой съёмки — не идеальное решение. Куда удобнее иметь архив, отсортированный по месту съёмки. Правда, место съёмки в EXIF не содержится. Таким образом, нужен некий интерфейс для классификации фотографий по месту съёмки. Эта задача тоже нашла простое решение. Для просмотра фотографий я использую программу gqview, в которой имеется возможность вызывать горячими клавишами до 10 разных редакторов для обработки выбранных файлов. Надо только написать соответствующий редактор, позволяющий указать место съёмки и установить его для заданных файлов. Написание такого редактора на языке командной оболочки оказалось элементарным:

Пример 9.1. Скрипт для сортировки фотографий, при необходимости получающий информацию из RAW

```
#!/bin/bash
NEFPATH=$NEFPATH./
for n in *.jpg;
  do
    date=`exif $n | grep 'Date and Time  '\
    | cut -d'|' -f2| cut -d' ' -f1| sed 's/:/-/g'`
   if [ $date ]; then true;
   else
        nef=`basename $n .jpg`
        nef=$NEFPATH$nef.nef
           if [ -r $nef ]; then
             exiftool -overwrite_original -TagsFromFile $nef $n
             echo $n
             date=`exif $n | grep 'Date and Time  '\
             | cut -d'|' -f2| cut -d' ' -f1| sed 's/:/-/g'`
             if [ -w $nef ];then
             mv $nef ../nef/
             fi
           fi
   fi
    if [ -L ../days/$date/$n ]; then true;
    else
        mkdir -p ../days/$date
        ln -s `pwd`/$n ../days/$date/$n
    fi
done
```

```
#!/bin/bash
location=`Xdialog --stdout --inputbox Location 0x0`
exiftool -overwrite_original -Location=$location $@
```

А вот сортирующий скрипт с поддержкой обработки поля `Location` (показан в примере 9.2). Есть в нём и ещё одно изменение: при сортировке создаются уменьшенные версии изображений и ссылки в каталогах, соответствующих разным местам съёмки, указывают именно на них, что ускоряет просмотр.

Пример 9.2. Скрипт с поддержкой сортировки по месту съёмки

```
#!/bin/bash
NEFPATH=$NEFPATH./

for n in *.jpg;
  do
    echo -n .
    date=‘exif $n | grep ’Date and Time  ’\
    | cut -d’|’ -f2| cut -d’ ’ -f1| sed ’s/:/-/g’‘
   if [ $date ]; then true;
   else
        nef=‘basename $n .jpg‘
        nef=$NEFPATH$nef.nef
           if [ -r $nef ]; then
             exiftool -overwrite_original -TagsFromFile $nef $n
             echo $n
             date=‘exif $n | grep ’Date and Time  ’\
             | cut -d’|’ -f2| cut -d’ ’ -f1| sed ’s/:/-/g’‘
             if [ -w $nef ];then
             mv $nef ../nef/
             fi
           fi
   fi
   if [ -r pics/$n ]; then true;
      else convert -resize 1024x768 -quality 90 $n pics/$n;
   fi

  location=‘exiftool -s -s -s -Location $n‘
  year_month=‘exiftool -s -s -s -d %Y-%m -CreateDate $n‘
    if [ $location != ’-’ ]; then
        if [ -L ../locations/$location/$year_month/$n ]; then true;
        else
                mkdir -p ../locations/$location/$year_month
                ln -s ‘pwd‘/pics/$n
../locations/$location/$year_month/$n
        fi
  fi
done
```


Всё бы хорошо, но работать этот скрипт стал ужасно медленно. Ещё бы — на обработку каждого изображения приходится 2–3 инициализации интерпретатора Perl (exiftool написан именно на нём) и масштабирование изображения. Впрочем, масштабирование производится для каждого файла только один раз, а вот 2–3 запуска exiftool производятся при каждом запуске скрипта. Через некоторое время мне надоело каждый раз ждать результатов обработки по несколько минут, и я предпринял очевидную оптимизацию: переписал скрипт на Perl. Заодно с оптимизацией я добавил в него довольно много новой функциональности: обработка нескольких JPEG-файлов, полученных из одного RAW (с именами `dsc_1234-1.jpg`, `dsc_1234-final.jpg` и т. п.), извлечение полноразмерного изображения низкого качества с целью отбора при помощи утилиты `nefextract` и др. Ниже приводится окончательный на сегодняшний день вариант скрипта. Он, разумеется, не является примером хорошего кода, но он решает свою задачу и экономит время. Хотя, возможно, со временем, я оптимизирую в нём хотя бы самые вопиющие места.

```
#!/usr/bin/perl

use strict;
use Image::ExifTool;

my $nefpath = $ENV{’NEFPATH’}?$ENV{’NEFPATH’}:’./’;

opendir DIR,’.’;
my @all = readdir(DIR);
my @nefs = grep { /\.(nef)$/ } @all;
closedir DIR;

foreach ( @nefs ) {
    my $name =$_;
    my $jpg=$name;
    $jpg =~ s/\.nef$/\.jpg/;
    if(! -r $jpg && ! -r $name.".jpg") {
            ‘nefextract $name > $name.jpg‘;
        print "$name\n";
            }
    }

opendir DIR,’.’;
@all = readdir(DIR);
```

```
my @files = grep { /\.(jpg)|(tif)$/ } @all;
closedir DIR;
if( my $mask=shift )
{
        @files = grep {/$mask/} @files;
}

foreach ( @files ) {
    my $name =$_;
    my $full_name = $name;
    my $info = Image::ExifTool::ImageInfo( $name );
    $name =~ /^([a-zA-Z]+_[0-9]+)/;
    my $nef = $nefpath.$1.'.nef';

    unless( $info->{'DateTimeOriginal'} ) {
        if( -r $nef ) {
            `exiftool -overwrite_original -TagsFromFile $nef
$name`;
            $info = Image::ExifTool::ImageInfo( $name );
            if ( -w $nef && !( $full_name =~ /nef/) ){
                `mv $nef ../nef/`;
            }
        }
        else
        {
            my $jpg = $nefpath.$1.'.jpg';
            if( './'.$name ne $jpg && -r $jpg )
            {
                `exiftool -overwrite_original -TagsFromFile $jpg
$name`;
                $info = Image::ExifTool::ImageInfo( $name );
            }
        }
    }
    if( -r $nef ) {
        if ( -w $nef && !( $full_name =~ /nef/) ){
            `mv $nef ../nef/`;
        }
    }

    unless ( -r  "pics/$name" ) {
        `convert -resize 1024x768 -quality 90 $name pics/$name`;
    }
```



```
    if( $info->{'DateTimeOriginal'} ) {
        my @ps = split /:/,$info->{'DateTimeOriginal'};
        my $y_m = $ps[0].'-'.$ps[1];
        my $loc =  $info->{'Location'};
        my $qual = $info->{'Quality'};
        my $pwd = `pwd`;
        $pwd =~ s/\n//;
        if( $loc ) {
            if( $qual eq 'Best' ) {
                unless( -l "../locations/$loc/$qual/$name") {
                    `mkdir -p ../locations/$loc/$qual/`;
                    `ln -s $pwd/pics/$name
../locations/$loc/$qual/$name`;
                }
            }
            unless( -l "../locations/$loc/$y_m/$name") {
                `mkdir -p ../locations/$loc/$y_m/`;
                `ln -s $pwd/pics/$name
../locations/$loc/$y_m/$name`;
            }
        }

    }
    print '.';
}
```

См. также Передача ввода/вывода программ в конвейере [стр. 112]

Самодельные мультфильмы

Антон Бояршинов

Идея эта посетила нас случайно и не помню уже как. С год назад мы с пятилетней дочерью уже обсуждали, как делают мультфильмы и нарисовали «мультфильм» на полях общей тетради и рисованный ролик секунд на 10 в XFig. Теперь же мы снимаем «кукольные» мультфильмы и это гораздо интересней. Отличное развлечение для детей и взрослых. «Кукольные» в кавычках потому, что часть мультфильмов, скорее «предметная», и чуть ли не лучшая часть.

В бытность мою школьником я занимался в детской киностудии и, в частности, именно мультипликацией. Насколько же более сложное дело это было тогда и насколько простое теперь! Цифровые технологии сделали съёмку простых мультфильмов доступной почти каждому, так как для этого пригоден практически любой цифровой фотоаппарат (даже практически не пригодный для фотосъёмки), и практически любой компьютер. А главное — результат виден практически сразу, и не надо ждать, пока кончится 30-метровая плёнка, чтоб увидеть, что же было снято за полгода. Короче, это здорово!

Съёмка

Вам понадобится цифровой фотоаппарат (практически любой) и штатив (хотя бы настольный или струбцина). Если снимать длинными зимними вечерами — без вспышки не обойтись. Я использую Nikon D70 и внешнюю вспышку, но, повторяю, аппаратура может быть практически любой. Разрешение и качество снимка я выставляю в минимальные значения (но учтите, что мои минимальные значения могут быть близки к вашим максимальным).

Всё просто. Сняли, подвинули, сняли, подвинули. В мультфильмах, которые мы снимаем с дочерью, частота смены кадров составляет от 4 до 6 кадров в секунду и это приемлемо, но учтите, что для плавности двигать надо совсем по чуть-чуть.

Сборка

Для сборки мультфильма потребуется программа `mogrify` из пакета ImageMagick и программа `mencoder` из одноимённого пакета.

Выгружаем всё в компьютер, складываем в отдельный пустой каталог. Для начала — уменьшаем. Я уменьшаю до размера 600×400 (у меня соотношение сторон $3 : 2$) вам может больше понравиться иной размер. Если снимки изначально маленького размера, например, 640×480, этот этап можно пропустить. Наиболее удобна для этого программа `mogrify`, которая может быть использована и для других пакетных операций над изображениями: вырезание фрагментов, добавление текста, рамок и тому подобное. В нашем случае командная строка может выглядеть так:

```
$ mogrify -resize 600x400 -format png *.jpg
```


Далее следует нарисовать титры соответствующих размеров и расположить их согласно порядку номеров файлов в нужном количестве.

Далее собственно сборка. Mencoder является программой кодирования видеопотоков и звука. Он может использовать в качестве исходного материала как различные видеоформаты, так и набор графических файлов. Именно последней возможностью мы и воспользуемся для создания мультфильма.

```
$ mencoder -o file.avi 'mf://*.png' -mf fps=4 -mf type=png -ovc
lavc
```

Где `fps=4` обозначает количество кадров в секунду, а `-ovc lavc` — метод кодирования изображения; всё, можно наслаждаться результатом: `mplayer -fs file.avi`

Как наложить звук?

```
$ mencoder -o file.avi 'mf://*.png' -mf fps=4 -mf type=png \
 -audiofile file.wav -oac mp3lame -ovc lavc
```

Где `file.wav` — это звук для мультфильма в форматах wav, mp3 или ogg/vorbis. Список поддерживаемых кодеков можно посмотреть, запустив `mencoder -oac help` или `mencoder -ovc help` соответственно.

Снимайте, смотрите, радуйтесь. Дети и жёны наверняка будут довольны. Некоторые наши мультфильмы можно скачать по адресу `ftp://ftp.altlinux.ru/pub/people/boyarsh/films/`.

См. также Средства работы с графикой, звуком и видео в Linux [снаружи, стр. 50]

Резервное копирование информации

Дмитрий Аленичев

В наше время резервное копирование информации необходимо хотя бы по той причине, что её потеря может сильно сказаться на времени, необходимом для восстановления испорченных или потерянных данных. Принятие простых мер поможет избежать необратимых последствий.

Хотя своевременное резервное копирование и не спасёт вас от всех возможных проблем с испорченными данными, оно, по крайней мере, сделает возможным восстановление сохранённой информации.

Создание резервных копий

Количество информации, которую вы сможете восстановить, во многом зависит от того, как часто вы её сохраняете и насколько надёжно храните.

Для начала определитесь с тем, где вы будите хранить backup-копии. Наилучшим решением для домашнего компьютера является использование отдельного жёсткого диска, но, к сожалению, не у всех есть такая возможность. Поэтому, скорее всего, вы будете сохранять копии в отдельной директории или на отдельном разделе жёсткого диска и периодически переписывать их на CD-R(W) или DVD-R(W).

Не используйте некачественные носители для сохранения резервных копий! Информация может быть дороже денег, затраченных на покупку качественных носителей.

Вне зависимости от того, где физически будет располагаться директория для сохранения копий, далее будем предполагать, что это директория `/backup`.

В простейшем случае скрипт для сохранение backup-копии будет выглядеть следующим образом:

```
#!/bin/sh
tar -zcf /backup/home.tar.gz /home
tar -zcf /backup/etc.tar.gz /etc
```

В этом скрипте нет ничего особенного, но основные принципы он демонстрирует:

- Сохранение и сжатие директории `/home` (домашние директории пользователей) в отдельный файл;
- Сохранение и сжатие директории `/etc` (общие системные настройки) в отдельный файл.

Рассмотрим использованные ключи команды tar:

`-z`	Сжать файл с использованием gzip
`-c`	Создать новый архив
`-f`	Использовать указанный файл

В случае порчи хотя бы нескольких байт сжатого резервного архива исключается возможность восстановления файлов из этого архива, даже если восстанавливаемый файл не находится в повреждённой области. Поэтому на такие носители, как магнитная лента, резервные копии предпочтительнее записывать в несжатом виде. Для этого не нужно указывать ключ `-z` команды tar.

Следующий скрипт реализует более широкие возможности для сохранения backup-копии:

```
#!/bin/sh
tar -zcvpf /backup/backup-‘date ’+%d-%B-%Y’‘.tar.gz \
--directory / --exclude=proc --exclude=var --exclude=mnt \
--exclude=usr --exclude=backup .
```

В данном примере сохраняются не отдельные директории, а корневая директория `/`, исключая `/proc`, `/var`, `/mnt`, `/usr` и, конечно, `/backup`. Также к имени файла добавляется дата создания резервной копии.

Дополнительно к уже рассмотренным были задействованы следующие ключи команды tar:

`-v`	Выводить список обработанных файлов
`-p`	Сохранять информацию о правах доступа
`--directory`	Директория для сохранения в архив
`--exclude`	Исключить директорию при сохранении в архив

Вы можете комбинировать оба предложенных варианта. Например, использовать первый скрипт ежедневно для ручного резервного копирования, а второй запускать по расписанию раз в неделю для полного сохранения.

Если вы решили использовать отдельный HDD для резервных копий, то вам подойдёт следующее решение.

Для начала подключите предварительно отформатированный HDD. Далее подразумевается, что это `/dev/hdb1`, т. е. первый раздел на втором диске канала IDE1. Создайте точку монтирования:

```
$ cd /
$ mkdir backup
```

Дополните ваши скрипты для резервного копирования следующими строками:

```
#!/bin/sh
mount /dev/hdb1 /backup
######################
# ваш скрипт         #
######################
umount /backup
```

Со временем вы сами разработаете наиболее удобный и эффективный для вас способ проведения регулярного резервного копирования. Например, вы можете исключить директории (ключ `--exclude`) с музыкой и фильмами из списка резервируемых директории.

Резервное копирование по расписанию

Далее вам предстоит настроить запуск скрипта по расписанию. На самом деле нет ничего проще. Со времён ОС UNIX существует программа `cron`, предназначенная для выполнения действий по расписанию. Откройте файл `/etc/crontab` и запишите новое правило:

```
#мин час  число  месяц  день недели  команда
0    1    *      *      5            /usr/bin/full-backup
```

Это правило будет выполняться каждую пятницу в 1 час ночи. Выберите удобное для вас время и запишите нужные значения в соответствующих колонках. В примере предполагается, что командой резервного копирования является `/usr/bin/full-backup`. Замените эту команду на имя вашего скрипта.

Отчёт о выполнении работы будет отправлен пользователю root по почте (при условии, что у вас настроена локальная доставка почтовых сообщений).

Восстановление из резервных копий

Ниже приведены основные команды для восстановления файлов из архива резервной копии. Перед замещением существующего файла убедитесь, что замена действительно необходима!

Перед извлечением файлов из резервной копии бывает необходимым просмотреть содержимое архива. Для этого укажите ключ `-t` команды tar. Например, следующая команда позволит просмотреть содержимое архива `/backup/backup-07-March-2005.tar.gz`:

```
$ tar -ztvpf /backup/backup-07-March-2005.tar.gz
```


Не забывайте о конвейерах. Для поиска файла в архиве вы можете использовать программу GNU grep:

```
$ tar -ztvpf /backup/backup-07-March-2005.tar.gz | grep smb.conf
```

Для извлечения файлов из архива предназначен ключ `-x` команды tar. Например, следующая команда восстановит все файлы из архива `/backup/backup-12-March-2005.tar.gz`:

```
$ tar -zxvpf /backup/backup-12-March-2005.tar.gz
```

Для восстановления определённых файлов из архива укажите их имена после имени архива. Например, следующая команда восстановит файлы `home/alenitchev/adt/backup.xml` и `etc/sendmail.cf` из архива `/backup/backup-17-March-2005.tar.gz`:

```
$ tar -zxvpf /backup/backup-17-March-2005.tar.gz \
home/alenitchev/adt/backup.xml etc/sendmail.cf
```

Перед восстановлением файла из резервной копии убедитесь, что восстанавливаемый файл не заменит более новый экземпляр.

Настройка почтовой системы в Linux

Дмитрий Аленичев

Введение

С каждым годом Интернет становится неотъемлемой часть жизни все большего количества людей. Одной из услуг, которые даёт нам глобальная сеть, является электронная почта. Для одних это средство ведения деловой переписки, для других возможность общения с людьми, находящимися в других странах. Но, как бы то ни было, реальности нашего времени диктуют свои правила. И сегодня электронный почтовый ящик есть даже у людей, отдалённо не представляющих, как устроена работа с почтой в операционных системах, созданных для работы в сети. Я, конечно, говорю о UNIX. А, если быть точным, о Linux, как наиболее используемом на рабочих станциях представителе POSIX-совместимых систем.

Из данного документа вы получите подробную информацию о настройке рабочей станции для работы с электронной почтой. Описаны все основные шаги, которые необходимо выполнить для создания

полнофункциональной почтовой системы. Представленные конфигурационные файлы помогут вам разобраться в настройке описанных программ.

Последнюю версию этого документа вы сможете найти по адресу: `http://gir.nongnu.org/people/daa/texts/linuxmail.html`. Если у вас есть чем дополнить этот документ, или вы нашли здесь информацию, на ваш взгляд, не соответствующую действительности, свяжитесь со мной по электронной почте: `alenitchev@nm.ru`. Спасибо всем, кто присылал комментарии, пожелания и вопросы.

Почтовая система

Привычное понятие «почта» складывается из нескольких совершенно разных действий: доставить письмо с почтового отделения (сервера) в ящик адресата, прочитать письмо и написать ответ, отправить письмо (отнести на почту или на сервер). В случае с бумажной почтой это делают разные люди (почтальон, разносящий почту по ящикам, сам адресат, почтальон, собирающий почту), естественно ожидать, что и в электронном варианте потребуется несколько программ.

Некоторые современные системы тяготеют к тому, чтобы втянуть максимум функций по работе с электронной почтой в интерфейс одной программы. Но традиционный подход UNIX, а за ней и Linux, заключается в том, что каждая программа должна выполнять только одну функцию, но делать это хорошо. В итоге это даёт пользователю гораздо больше гибкости в настройке окружения «под себя».

Почтовое сообщение

Перед чем говорить о настройке почтовой системы, нужно уяснить, что представляет собой почтовое сообщение. Почтовое сообщение состоит из *заголовка* (служебная информация, адрес отправителя, получателя и т. д.) и *тела* (текста, написанного отправителем). Заголовок отделяется от тела пустой строкой. Попробую объяснить, что могут включать в себя заголовок и тело письма.

Заголовок письма

Здесь представлены некоторые поля заголовка (см. таблицу 9.1). Не все поля являются обязательными для заполнения. Вам, вероятно, будет достаточно заполнить поля «To», «From» и, возможно, «Subject».

Поле	Назначение
`To`	Адрес получателя. Если указано несколько адресов, то они разделяются запятыми
`Cc`	Адрес получателя копии сообщения. Если указано несколько адресов, то они разделяются запятыми. Получатели «Cc»-копии увидят адреса других получателей копии
`Bcc`	Адрес получателя невидимой копии. Если указано несколько адресов, то они разделяются запятыми. Отличается от «Cc» тем, что получатели «Bcc»-копии не увидят адресов других получателей копии
`From`	Адрес отправителя
`Subject`	Тема сообщения
`Date`	Дата отправления сообщения
`Reply-To`	Адрес для ответа
`Message-ID`	Автоматически генерируемая строка
`Received`	При прохождении сообщения через каждый пункт, который необходимо пройти для доставки, вставляется данная строка, в которой указано имя пункта, время и дата получения сообщения, откуда оно было получено этим пунктом, идентификатор сообщения, какое транспортное программное обеспечение использовалось. По этим заголовкам вы сможете проследить путь сообщения
`X-*`	Заголовки, начинающиеся с X-, вы можете определять сами и заносить в них любую информацию
`Organization`	Название организации отправителя
`User-Agent`	Почтовый клиент, с помощью которого отправитель создал текст письма

Таблица 9.1. Поля заголовка письма

Тело письма

В теле письма содержится набранный вами текст. Все вложения, которые вы определили для добавления в письмо, также будет помещены в тело.

Получение почты

Задачи программы доставки почты заключаются в следующем:

- Получение почты с почтового сервера, на котором расположен ваш почтовый ящик, на ваш компьютер. Доставка может осуществляться по нескольким почтовым протоколам, наиболее распространённые — POP3 и IMAP.
- Передача писем программе обработки почты — для помещения в файл почтового ящика пользователя (возможно, с сортировкой по нескольким ящикам).

Рассмотрим две программы данного типа: fetchmail и getmail. Они очень сильно различаются по возможностям. Fetchmail работает с большим количеством протоколов (POP2, POP3, RPOP, APOP, KPOP, IMAP4 и другие) и имеет огромное количество различных функций. Getmail, наоборот, работает только с POP3, но многим она больше нравится за свою простоту.

fetchmail

Сайт проекта: `http://www.catb.org/~esr/fetchmail`

Чтобы получать сообщения с почтового сервера, нужно сообщить программе получения почты сведения о вашем почтовом ящике: адрес сервера, название учётной записи (часть адреса до «`@`»), пароль, способ доступа (протокол). В fetchmail наиболее эффективно указать все эти данные в конфигурационном файле в домашнем каталоге пользователя. Создайте файл `~/.fetchmailrc` (см. пример 9.3):

```
[user@machine ~/]$ touch ~/.fetchmailrc
```

Измените права доступа, т. к. в этом файле будут храниться ваши пароли от почты, и он не должен быть доступен для чтения никому, кроме вас и запущенной вами программе fetchmail:

```
[user@machine ~/]$ chmod 0600 ~/.fetchmailrc
```


Пример 9.3. Пример конфигурационного файла .fetchmailrc

```
# .fetchmailrc - конфигурационный файл для fetchmail
# записывать события в системный журнал
set syslog
# общие для всех учётных записей настройки
defaults    protocol pop3,  # протокол
            timeout 60,     # время ожидания (в секундах)
            nokeep,         # полученные письма удалять с сервера
            fetchall        # получать всю почту
# специфичные для учётных записей настройки (сервер, login и
пароль)
#
# Учётная запись 1
poll "pop.mailhost.ru",
        user "username",
        password "pass";
#
# Учётная запись 2
poll "mail.freemail.ru",
        user "username",
        password "pass";
```

Таким же образом опишите в конфигурационном файле все свои почтовые ящики, и можно выполнять команду `fetchmail` для получения почты. Естественно, к моменту запуска fetchmail соединение с Интернет должно быть уже установлено, fetchmail не предназначен для установки соединения и сразу обращается к сети. Fetchmail станет получать почту последовательно из всех почтовых ящиков, перечисленных в конфигурационном файле. Примеры более сложной конфигурации можно найти в документации, сопровождающей fetchmail.

После того как сообщения получены с почтового сервера, необходимо доставить их непосредственно в файл почтового ящика пользователя. В полном соответствии с принципом «одна программа — одна функция», fetchmail доставкой самостоятельно не занимается, сразу передавая полученную почту специально предназначенной для этого программе. Однако здесь есть некоторые альтернативы.

По умолчанию fetchmail передаст полученные письма службе передачи сообщений (MTA, Mail Transport Agent), в Linux-системах в таком качестве выступает локальный SMTP-сервер. Такая служба по-

чти всегда присутствует в системе, однако в некоторых ситуациях, например, если для отправки почты вы пользуетесь программами-дополнениями к почтовому клиенту, она может и отсутствовать. В этом случае пригодится опция, указывающая fetchmail передавать полученную почту программе доставки сообщений (MDA, Mail Delivery Agent). Для этого в приведённом выше примере конфигурационного файла следует поставить запятую после опции `fetchall` и указать опцию `mda procmail -f %F`:

```
...
fetchall,      # получать всю почту
mda procmail -f %F
...
```

В данном случае в качестве MDA выступает программа для сортировки сообщений procmail, а наличие SMTP-сервера вообще не требуется. Хотя я не считаю отказ от установки SMTP-сервера хорошей идеей. Но если SMTP-сервер установлен, то большой разницы между обработкой почты с помощью MTA или MDA нет, т. к. практически каждый MTA вызовет procmail для обработки почты.

getmail

Принцип работы с getmail такой же, как и с fetchmail: сначала нужно описать все свои почтовые ящики в конфигурационном файле. Создайте директорию `~/.getmail/` и файл `getmailrc` в ней:

```
[user@machine ~/]$ mkdir ~/.getmail
[user@machine ~/]$ touch ~/.getmail/getmailrc
```

Точно так же опишите все свои почтовые ящики (см. пример 9.4) и запускайте команду `getmail`. Подробности, как обычно, можно найти в документации.

Обработка почты

Сайт проекта: `http://www.procmail.org`

Путь письма заканчивается в одном или нескольких файлах, которые служат почтовыми ящиками и архивами, где накапливаются письма. Существует несколько форматов таких файлов, но я хочу заметить, что если вы пользуетесь Linux, то корреспонденцию предпочтительнее хранить в формате mbox. Этот формат представляет собой

Пример 9.4. Пример конфигурационного файла getmailrc

```
# getmailrc - конфигурационный файл для getmail
# общие для всех учётных записей настройки
[default]
message_log = "~/.getmail/log"        # путь к логу
delete = 1                          # удалять почту на сервере
после получения
readall = 1                          # получать всю почту
postmaster = "| procmail"         # почту на обработку
#
# Учётная запись 1
[username_at_mailhost]
server = pop.mailhost.ru
username = username
password = "pass"
#
# Учётная запись 2
[username_at_freemail]
server = mail.freemail.ru
username = username
password = "pass"
```

обычный текстовый файл, в котором сообщения записываются одно за другим: сначала заголовки сообщения (в виде обычного текста, по одному полю заголовка на строку), затем его тело. С таким почтовым ящиком можно работать как с обычным текстом при помощи стандартных утилит. Рассмотрим небольшой пример.

Допустим, что вам нужно составить список всех e-mail адресов отправителей, взятых из заголовков писем, находящихся в вашем рабочем почтовом файле. Затем этот список необходимо отсортировать в алфавитном порядке, удалить дубликаты и записать в файл. В случае с mbox это очень просто. Достаточно всего одной строчки:

```
[user@machine ~/]$ cat ~/Mail/work | grep '^From: ' | sort | uniq
> list
```

Однако хранить все письма в одном файле неудобно. Намного эффективнее сортировать сообщения от различных корреспондентов и из разных списков рассылки по отдельным файлам. Автоматизировать задачу сортировки почты по разным критериям вам поможет procmail.

Пример 9.5. Пример конфигурационного файла .procmailrc

```
# .procmailrc - конфигурационный файл для procmail
# ----------
# Переменные
# ----------
MAILDIR=~/Mail                    # каталог с почтой
LOGFILE=~/Mail/procmail.log  # журнал для записи всех операций
DEFAULT=~/Mail/mbox          # файл для неотсортированных
сообщений
# ------------------------
# Правила сортировки почты
# ------------------------
# Пример: сортировка почты, приходящей на e-mail user_name@host.ru
:0
* ^To.*user_name@host.ru       # условие (To: user_name@host.ru)
host_mail                      # файл host_mail в каталоге MAILDIR

# Пример: mailing-list Moscow LUG
:0
* ^To.*mlug@UnixCenter.RU      # условие (To: mlug@UnixCenter.RU)
mlug                           # файл mlug в каталоге MAILDIR

# Пример: сортировка от адресатов из домена friends.org
:0
* ^From.*@friends.org          # условие (From: *@friends.org)
friends                        # файл friends в каталоге MAILDIR

# Пример: удаление писем от адресата spammer@mail.ru
:0
* ^From.*spammer@mail.ru       # условие (From: spammer@mail.ru)
/dev/null                      # выбросить -- /dev/null

# Пример: автоматическая отсылка PGP-ключа
0:
* ^Subject.*PGP                # условие (Subject: PGP)
| (formail -r ; cat ~/key.asc) # вывести PGP-ключ
| sendmail -t                  # отправить его
```


Procmail будет раскладывать вашу корреспонденцию по отдельным почтовым файлам на основании содержания тех или иных полей заголовка письма. Принцип работы с procmail всё тот же: правила для сортировки сообщений нужно описать в конфигурационном файле. Создайте файл `~/.procmailrc` (см. пример 9.5):

```
[user@machine ~/]$ touch ~/.procmailrc
```

В этом примере приведены самые простые, но и самые нужные виды правил сортировки. Procmail позволяет писать правила практически неограниченной сложности — всё зависит от ваших потребностей. Подробную информацию и примеры можно найти в руководствах procmailrc(5) и procmailex(5).

Если в вашем почтовом файле *уже* находится большое количество писем, нуждающихся в сортировке, то вы можете воспользоваться следующей командой:

```
[user@machine ~/]$ cat ~/Mail/mbox | formail -s procmail
```

Просмотр и написание почты

Сайт проекта: `http://www.mutt.org`

Почтовые клиенты (MUA, Mail User Agent) — программы для чтения и написания почты — многочисленны и многообразны. К основным простым функциям чтения/написания могут быть добавлены самые разнообразные дополнительные удобства. В качестве MUA я предлагаю использовать mutt. Комментарии, я думаю, излишни, т. к. это самая удобная программа для чтения и написания писем. Попробую коротко описать, как её настраивать и использовать.

Все настройки mutt описываются в конфигурационном файле `~/.muttrc`:

```
# .muttrc - конфигурационный файл для mutt
# ---------------
# Общие настройки
# ---------------
set folder=~/Mail               # каталог для почтовых ящиков
set alias_file=~/.mail_aliases  # файл алиасов
set arrow_cursor                # курсор в виде '->'
set attribution="%d, %n написал:"
                                # начало письма при ответе
```

```
set copy=yes                    # сохранять копии исходящих писем
set edit_headers                # редактировать заголовки письма
set editor="vim"                # редактор писем
set folder_format="%t%N %-30.30f %8s"
                                # вид списка папок
set index_format="%4C %Z %{%b %d} %-31.31F %N (%4c) %s"
                                # вид списков писем
set mailcap_path="~/.mailcap"   # путь к .mailcap
set menu_scroll                 # прокручивать список по одной
строке
set mail_check=5                # интервал проверки почты
set mbox=+mbox                  # файл для прочитанных сообщений
set mime_forward=ask-no         # запрашивать вид пересылаемого
письма
set pager_stop                  # в конце письма не переходить на
следующее
set postponed=+drafts           # черновики
set print=ask-yes               # запрос перед печатью файла
set print_command=lpr           # команда печати
set record=+sent                # куда сохранять отправленные
письма
set signature="~/.signature"    # файл с подписью к сообщениям
set show_alt                    # y/n вместо д/н
set send_charset="us-ascii:iso-8859-1:koi8-r:windows-1251:utf-8"
                                # кодировка отправляемых писем
set visual=vim                  # редактор, вызываемый по ~v

# сортировка писем при просмотре почтового ящика
set sort=threads
set sort_aux=reverse-date-received
set sort_browser=reverse-date

# путь к спулу почты
# procmail уже обработал почту, и вы можете указать здесь файл
mbox
set spoolfile='~/Mail/mbox'

# поле From заголовка отправляемых писем
set from="Your Name <user_name@host.domain>"

# собственные заголовки писем можно устанавливать с помощью my_hdr
my_hdr X-Url: http://alenitchev.nm.ru
```



```
# ------------------------------------
# Настройки просматриваемых заголовков
# ------------------------------------
ignore *
# поля, которые я хочу видеть
unignore        from: subject to cc mail-followup-to \
                date x-mailer x-url user-agent reply-to

# ---------------
# Привязки клавиш
# ---------------

# клавиша <up> при просмотре письма на строку вверх
bind pager <up> previous-line
# клавиша <down> при просмотре письма на строку вниз
bind pager <down> next-line

# --------------
# Почтовые файлы
# --------------

# перечислите все ваши почтовые файлы
mailboxes +mbox +work +friends +lists/debian +lists/mlug

# если вы подписаны на списки рассылки, укажите их адреса
# это нужно для использования возможности list-reply (клавиша 'L')
lists mlug@unixcenter.ru debian-russian@lists.debian.org

folder-hook mlug "push ^[V"     # при открытии этих файлов
выполнять
folder-hook debian "push ^[V"   # 'Esc-V' (свернуть все дискуссии)

# при открытии mlug устанавливать заголовок Reply-To
folder-hook mlug "my_hdr Reply-To: \
Moscow Linux User Group <mlug@unixcenter.ru>"

# при открытии debian устанавливать заголовок Reply-To
folder-hook debian "my_hdr Reply-To: \
Debian <debian-russian@lists.debian.org>"

# -------------------
# Алиасы (псевдонимы)
# -------------------
```

```
# Очень удобное средство -- укажите здесь псевдонимы и адреса
# получателей. Для написания письма можно будет использовать
# команду mutt alias_name. Адрес, соответствующий
# alias_name, будет подставлен в поле From.
alias mlug Moscow Linux User Group <mlug@unixcenter.ru>
alias support TechSupport <support@hostname.ru>
```

В директории `/etc` находится файл `Muttrc`. Он может послужить вам хорошим примером конфигурационного файла для mutt — к каждому параметру прямо в этом файле дан краткий комментарий: смысл параметра, значение по умолчанию. Кроме того, в пакете mutt распространяется несколько примеров конфигурационного файла mutt для специальных задач, например, для работы с электронными подписями (PGP).

Важное удобство при работе с электронной почтой — возможность просматривать вложения (аттачменты) в любых форматах прямо из почтового клиента. В mutt для этого предусмотрена возможность вызова внешней программы для просмотра вложения. Какую именно программу вызывать, определяется по MIME-типу вложения. Связь между MIME-типами и программами для просмотра устанавливается в файле `~/.mailcap`. Приведу небольшой пример, чтобы продемонстрировать структуру этого файла (см. пример 9.6).

Пример 9.6. Пример конфигурационного файла .mailcap

```
# .mailcap - обработка MIME-типов в mutt
image/*; ee %s
audio/mod; mikmod %s
text/html; links -g %s
application/pdf; xpdf %s
```

Если в вашей системе установлен пакет mailcap, то не обязательно самостоятельно задавать все привязки: mutt умеет пользоваться общесистемным mailcap (обычно `/etc/mailcap`). В своём файле `~/.mailcap` можно будет переопределить привязки, перечислив только те типы, обработку которых нужно изменить. Все остальные типы будут попрежнему обрабатываться в соответствии с общесистемным `mailcap`. Думаю, что те типы, которые вам хотелось бы обрабатывать, вы без труда добавите сами, основываясь на вышеизложенном примере.

Последний штрих в настройке mutt — создать файл подписи. Файл подписи (по умолчанию `~/.signature`) может содержать любой текст, который будет добавлен к вашим исходящим письмам. Mutt позволяет

Клавиша	Описание
`c`	Адресаты CC (копии письма)
`b`	Адресаты BCC (слепые копии)
`s`	Редактировать поле Subject
`r`	Редактировать поле Reply-To
`a`	Прикрепить файл в виде вложения
`e`	Вернуться к редактированию сообщения
`C`	Копировать сообщение в файл
`q`	Отложить сообщение
`y`	Отправить сообщение
`?`	Просмотреть справку

Таблица 9.2. Клавиши mutt. Создание сообщения.

Клавиша	Описание
`Вверх`	Прокрутка письма вверх на одну строчку
`Вниз`	Прокрутка письма вниз на одну строчку
`q`	Закончить чтение
`?`	Просмотреть справку

Таблица 9.3. Клавиши mutt. Чтение сообщения.

иметь несколько разных подписей, подставляя нужную в зависимости от адресата или других параметров. Подробности можно найти в документации по mutt.

Итак, настройка окончена, можно запускать:

```
[user@machine ~/]$ mutt
```

Перед собой вы увидите список писем из файла `mbox` (`~/Mail/mbox`). Концепция работы в mutt состоит в том, что каждая функция вызывается нажатием по возможности одной клавиши, обычно буквенной, причём такой, чтобы буква напоминала о самой функции. Так, чтобы написать письмо, нужно нажать «`m`» — от mail. В самой верхней

Клавиша	Описание
Home	Перейти к первому сообщению
End	Перейти к последнему сообщению
PageUp	Перейти вверх на один экран
PageDown	Перейти вниз на один экран
Вверх	Перейти к предыдущему сообщению
Вниз	Перейти к следующему сообщению
Enter	Чтение письма
c	Открыть почтовый файл. Внизу экрана вы увидите предложение ввести имя файла. Нажатием Tab или ? вы откроете меню выбора файла.
m	Mail. Написать новое письмо
r	Reply. Ответить на текущее сообщение
f	Forward. Переслать текущее сообщение
d	Delete. Удалить текущее сообщение
C	Copy. Копировать текущее сообщение в другой файл
L	List-reply. Ответить на сообщение в список рассылки
q	Quit. Выйти из программы
?	Просмотреть справку

Таблица 9.4. Клавиши mutt. Главное окно.

строке экрана в строке подсказок перечислены клавиши для основных функций, доступный в текущий момент (см. таблицы 9.2, 9.3, 9.4).

После нажатия на клавишу *m* (написать новое письмо) будет открыт текстовый редактор, в котором вы сможете набрать текст почтового сообщения. В mutt есть свой собственный текстовый редактор, однако более удобно использовать для редактирования писем тот редактор, к которому вы привыкли и которым редактируете любые текстовые файлы. Внешний редактор для написания писем определяется в файле `~/.muttrc` параметром `set editor=`. После выхода из редактора появится окно mutt, в котором вы сможете произвести ряд действий перед отправкой письма.

Если новая почта у вас проходит сортировку и приходит в несколько разных ящиков, то удобно запускать mutt с ключом `-y`. В этом случае mutt вместо списка писем в ящике отобразит список почтовых ящиков. Все ваши ящики нужно перечислить в `~/.muttrc`: `mailboxes +mbox +work +friends`. Если в каком-то из них есть новая почта, в списке будет стоят флаг «N», перейти внутрь ящика можно клавишей *Enter*. По команде `mutt -f filename` mutt сразу откроет указанный почтовый ящик.

Отправка почты

Чтобы отправить письмо, нужно установить соединение с почтовым сервером адресата и передать ему сообщение по протоколу SMTP. Можно не обращаться к серверу адресата напрямую, а воспользоваться пересылкой (relay), передав сообщение любому серверу, который согласится его у вас принять, а уж он сам позаботится о его доставке до почтового сервера адресата. Для пересылки сообщений (организации SMTP-серверов) предназначены MTA, существует несколько распространённых программ этого типа. Я расскажу о настройке некоторых из них.

При настройке службы MTA нужно позаботиться о том, чтобы ваш SMTP-сервер не принимал сообщения для пересылки ни от кого, кроме локальных пользователей (запретить relay). В противном случае ваш SMTP-сервер могут использовать для пересылки спама. В большинстве современных дистрибутивов любой MTA распространяется с настройками по умолчанию, запрещающими открытый relay, но изменяя настройки вручную, нужно следить за тем, чтобы этот запрет оставался в силе.

У вас есть выбор: организовать свой собственный (локальный) SMTP-сервер, который будет обращаться напрямую к почтовым серверам адресата, или использовать какой-нибудь внешний сервер (например, провайдера). Во втором случае в настройках MTA нужно указать адрес внешнего сервера, в разных программах такой параметр называется smarthost или relayhost. При использовании внешнего SMTP-сервера можно обойтись вообще без MTA, отправляя почту с помощью программы-расширения для почтового клиента (об этом см. ниже).

Для работы MTA необходимо, чтобы было правильно установлено полное имя машины (FQDN, fully qualified domain name). В разных дистрибутивах Linux процедура настройки сети может быть устроена несколько по-разному, поэтому за инструкциями по настройке пол-

ного имени обращайтесь к документации по вашему дистрибутиву. Проверить, что полное имя настроено правильно, можно командой `hostname -f`, которая должна вернуть полное имя (вместе с доменом).

```
[user@machine ~/]$ hostname -f
yourhostname.yourdomainname
```

Sendmail

Сайт проекта: `http://www.sendmail.org`

Sendmail — это дедушка всех MTA, первый и в течение некоторого времени единственный SMTP-транспорт. До настоящего времени он остаётся довольно распространённым на почтовых серверах в Интернет и предлагается в качестве MTA по умолчанию в некоторых дистрибутивах Linux.

Основной конфигурационный файл sendmail, `sendmail.cf`, прославлен своей сложностью и для неподготовленного читателя представляет собой просто абракадабру. Для упрощения настройки было создано несколько более высокоуровневых средств, таких как макроконфиги (`sendmail.mc`) и утилиты для автоматической генерации конфигурационного файла — конфигураторы (`sendmailconfig`).

Настройка smarthost Для пересылки писем через внешний сервер потребуется указать smarthost. Как это сделать с помощью конфигуратора или макроконфига, описано в документации к ним. Есть способ сделать это наверняка, напрямую отредактировав `sendmail.cf`. Откройте `/etc/sendmail.cf` и найдите в начале файла строки:

```
# "Smart" relay host (may be null)
DS
```

Вот после этого DS и запишите в квадратных скобках имя smtp-сервера. Должно получиться вот так:

```
# "Smart" relay host (may be null)
DS[smtp.server.ru]
```

Теперь письма будут отправляться через указанный smtp-сервер. Недостаток этого метода в том, что при этом вы лишаетесь некоторых гарантий целостности и непротиворечивости настроек, которые обеспечивает высокоуровневый конфигуратор.

Ограничение доступа При настройке SMTP-сервера обязательно обратите внимание на ограничения доступа к вашему серверу — по уже названным выше причинам не следует делать его по умолчанию доступным для всех. Наилучшим решением будет запрет доступа для всех, кроме локальных пользователей.

Sendmail позволяет организовать и более гибкую политику доступа, устанавливая выборочные ограничения. Для этого вы можете использовать следующую последовательность действий. Добавьте в файл `/etc/mail/access` имя домена и тип ограничения:

```
spamers.localdomain        550 Spam is bad!
dial-up_hosts.domain        REJECT
adsl_hosts.domain         REJECT
```

Чтобы изменения вступили в силу, необходимо обновить файл `/etc/mail/access.db`:

```
[root@machine ~/]# /usr/sbin/makemap hash /etc/mail/access.db < \
/etc/mail/access
```

...и перезапустить службу sendmail:

```
[root@machine ~/]# /etc/rc.d/init.d/sendmail restart
```

Приведённый пример служит только для демонстрации возможностей, полные и более конкретные сведения нужно искать в документации по sendmail.

Управление почтовой очередью Почтовая очередь хранится в директории `/var/spool/mqueue`. Для просмотра почтовых сообщений, находящихся в очереди, а также их состояния вы можете воспользоваться программой `mailq`:

```
[user@machine ~/]$ mailq
/var/spool/mqueue (1 request)
----Q-ID---- --Size-- -----Q-Time-----
---------Sender/Recipient---------
j1RIZop10488       59 Sun Feb 27 21:35 alenitchev
                 (host map: lookup (fake.net): deferred)
                                     fake@fake.net
```

Как видно из приведённого выше примера, адрес хоста fake.net не найден, и поэтому сообщение оставлено в очереди в статусе ожидания. Попытка отправки сообщения будет повторена при следующей обработке очереди. Для немедленной обработки почтовой очереди предназначена команда `runq`.

Exim

Сайт проекта: `http://www.exim.org`

Exim значительно дружелюбнее sendmail'а в плане настройки. Особенно проста настройка exim в дистрибутиве Debian GNU/Linux, где она является MTA по умолчанию.

Вы можете настроить exim во время этапа базовой настройки Debian или сделать это в любое время с помощью программы `eximconfig`. Рассмотрим конфигурацию для отправки почты через внешний сервер (smarthost).

В меню `eximconfig` выберите пункт «mail sent by smarthost; received via SMTP or fetchmail», введите почтовое имя машины и настройте параметры, связанные с приёмом почты от других компьютеров по своему усмотрению. Наконец, введите имя сервера, выступающего в роли smarthost.

Теперь вам нужно настроить перезапись адреса. Добавьте в файл `/etc/exim/exim.conf` секцию «`REWRITE CONFIGURATION`»:

```
*@localhost       ${lookup{$1}lsearch{/etc/email-addresses}\
                                        {$value}fail} Ffsr
```

Для окончания настройки перезаписи адреса добавьте в файл `/etc/email-addresses` записи для пользователей:

```
root:  me@somewhere.tut
me:    me@somewhere.tam
metoo: me@anywhere.else
```

Esmtp и Msmtp

В этом разделе описаны программы, не являющиеся MTA, но тем не менее позволяющие отправлять почту. Такой тип программ называется программами-дополнениями к почтовому клиенту («smtp-plugins for MUA»), то есть надстройка над MUA, позволяющая отсылать корреспонденцию через внешний smtp-сервер. Если ваша задача ограничивается отсылкой почты через внешний сервер, то такие программы будут хорошим решением: они предельно просты в настройке (см. примеры 9.7 и 9.8) и удобны в использовании.

Эти программы позволят вам отправлять почту через внешний smtp-сервер, не забивая себе голову проблемами администрирования собственного почтового сервера.

Пример 9.7. Пример конфигурационного файла .esmtprc

```
# .esmtprc - конфигурационный файл для esmtp
#
hostname = smtp.mail_host.ru:25
username = "user_name"
password = "password"
```

Пример 9.8. Пример конфигурационного файла .msmtprc

```
# .msmtprc - конфигурационный файл для msmtp
# --------------------------
# Учётная запись по умолчанию
# --------------------------
account default
host smtp.work_host.ru       # smtp-сервер
from login@work_host.ru      # e-mail
user login                   # логин
password pass                # пароль
# ------------------------
# Локальная учётная запись
# ------------------------
account local                # имя учётной записи
host localhost               # используется локальный smtp-сервер
from user_name@hostname.ru   # e-mail
# ---------------------------------------------------------------
# Учётная запись провайдера с аутентификацией CRAM-MD5. Порт 2500
# ---------------------------------------------------------------
account provider             # имя учётной записи
host mail.super_provider.ru  # smtp-сервер
port 2500                    # порт
from login@super_provider.ru # e-mail
auth cram-md5                # аутентификация CRAM-MD5
user login                   # логин
password "pass"              # пароль
```

esmtp

Сайт проекта: `http://esmtp.sourceforge.net`

Чтобы использовать esmtp для отсылки сообщений, в `~/.muttrc` необходимо добавить следующую строчку:

```
set sendmail="esmtp -v"
```

msmtp

Сайт проекта: http://msmtp.sourceforge.net

В ~/.muttrc необходимо добавить следующую строчку:

```
set sendmail="msmtp"
```

Другие программы

Помимо перечисленных в этом разделе программ, существует огромное количество других MTA. Наиболее известные среди оставшихся — qmail и postfix. Традиционно считается, что они больше подходят для использования на сервере, чем на локальной машине. Информацию о их настройке и использовании вы найдёте в документации в соответствующих пакетах и в Интернет.

Адресная книга

Сайт проекта: http://abook.sourceforge.net

Итак, ваша система уже позволяет получать, сортировать, читать, писать и отправлять письма. Теперь пора позаботиться об удобстве хранения многочисленных адресов получателей ваших сообщений. Я расскажу про программу abook, наиболее удобную консольную адресную книгу.

Запускайте abook:

```
[user@machine ~/]$ abook
```

Разобраться в интерфейсе этой программы не составит труда.

Приведу пример добавления записи. Нажимайте клавишу *a* и вводите имя. Открывается окно просмотра и редактирования записи. Клавиши *2*, *3*, *4*, *5* служат для добавления к записи e-mail адресов. С помощью клавиш *Влево*, *Вправо* вы можете перемещаться между разделами «CONTACT», «ADDRESS», «PHONE», «OTHER». Окончив редактирование записи нажатием на клавишу *Enter*, возвращайтесь к списку записей. Для написания письма выбранному адресату предназначена клавиша *m*. Если вы хотите написать письмо на несколько адресов сразу, то выделите нужные записи нажатием пробела и нажмите *m*. В abook, так же как и в mutt, имеется встроенная справка — *?*.

Записи вашей адресной книги хранятся в файле ~/.abook.addressbook, а настройки программы — в ~/.abookrc.

У этой программы имеется одна полезная возможность, которая может вам пригодиться. Это конвертирование файлов, содержащих адресную книгу, между различными форматами (см. таблицы 9.5, 9.6). Например, вам нужно получить HTML-версию вашей адресной книги. Для этого вы можете воспользоваться следующей командой:

```
[user@machine ~/]$ abook --convert abook .abook.addressbook html \
addressbook.html
```

Синтаксис этой команды:

```
abook --convert ФорматВходногоФайла ВходнойФайл
ФорматВыходногоФайла \
ВыходнойФайл
```

Имя	Описание
abook	«родной» формат программы abook
ldif	адресная книга ldif / Netscape
pine	адресная книга pine
csv	comma separated values

Таблица 9.5. Форматы входного файла abook

Имя	Описание
abook	«родной» формат программы abook
mutt	mutt alias
html	HTML-файл
pine	адресная книга pine
gcrd	адресная книга GnomeCard (VCard)
csv	comma separated values
elm	elm alias
text	текст

Таблица 9.6. Форматы выходного файла abook

Шифрование писем

Сайт проекта: `http://gnupg.org`

Для начала вам необходимо запустить gpg без параметров для создания каталога `~/.gnupg`, в котором будут храниться различные конфигурационные файлы:

```
[user@machine ~/]$ gpg
gpg: /home/alenitchev/.gnupg: directory created
gpg: /home/alenitchev/.gnupg/options: new options file created
gpg: you have to start GnuPG again, so it can read the new options
file
```

Теперь вы можете создать пару ключей (публичный и секретный) для шифрования писем и файлов:

```
[user@machine ~/]$ gpg --gen-key
```

Вам потребуется ответить на несколько вопросов. На этом этапе затруднений у вас возникнуть не должно. Отмечу лишь, что в качестве ключевой фразы (passphrase) необходимо выбрать что-нибудь посложнее. После окончания генерации ключа вы можете использовать GnuPG.

Экспорт открытого ключа

Экспортировать свой открытый ключ вы можете следующим образом:

```
[user@machine ~/]$ gpg --export -a Dmitri Alenitchev >
gpg-public-key.asc
```

Этой командой вы экспортируете открытый ключ пользователя Dmitri Alenitchev и перенаправите вывод в файл `gpg-public-key.asc`.

Импорт открытого ключа

Импортировать чей-нибудь открытый ключ позволяет команда:

```
[user@machine ~/]$ gpg --import gpg-public-key.asc
```

Например:


```
[user@machine ~/]$ gpg --import friend-key.asc
gpg: key 8421F11C: public key imported
gpg: Total number processed: 1
gpg:               imported: 1
[user@machine ~/]$ gpg --list-keys
/home/alenitchev/.gnupg/pubring.gpg
------------------------
pub  1024D/F18D5DDB 2004-12-11 Dmitri Alenitchev
<alenitchev@nm.ru>
sub  2048g/8E70D455 2004-12-11 [expires: 2005-12-11]

pub  1024D/8421F12C 1997-10-05 My Friend <friend@host.domain>
sub  2048g/5E985ED4 1997-10-05
```

Справку о других командах вы можете получить, запустив gpg с ключом `--help`

Перед тем как отправить кому-нибудь зашифрованное письмо, вам необходимо импортировать публичный ключ адресата!

GnuPG + mutt

Теперь пора поговорить об использовании GnuPG при отправке и получении почты. Создавайте новое письмо в mutt, перед отправкой нажимайте клавишу *p*. Внизу экрана вы увидите приглашение:

```
(e)шифр, (s)подпись, (a)подпись как, (b)оба, (f)отказаться?
```

Зашифруйте своё сообщение и отправляйте адресату. После того как вы нажмёте *y* для отправки письма, вам будет предложено ввести идентификатор ключа. Вводите имя адресата и выбирайте ключ из списка.

Защита от спама

О том, что такое спам, и зачем от него защищаться, наверняка все слышали. Здесь вы не найдёте описания специальных программ, предназначенных для защиты от спама, оставим это для администраторов почтовых серверов. Я расскажу про то, как можно фильтровать сообщения, основываясь на так называемых «белых» и «чёрных» списках. Этот способ поможет снизить количество спама, которое вы видите перед собой ежедневно. Приступим.

Добавление адресов в «чёрный» или «белый» список

Для начала напишем скрипт, который будет извлекать почтовый адрес из поля From в заголовке письма.

```
#!/usr/bin/perl -wn
unless (/\</) { print; } else { print /<([^>]+)/, "\n"; }
```

Назовите его `listadd`, поместите в директорию `/usr/local/bin` и выполните команду:

```
[root@machine ~/]# chmod +x /usr/local/bin/listadd
```

Добавьте к вашему `~/.muttrc` следующие строки:

```
# ---------- Add to black/white list ---------- #
# Для добавления адресата из режима списка писем
macro index \eb '| formail -x From: | listadd >> ~/Mail/blacklist'
macro index \ew '| formail -x From: | listadd >> ~/Mail/whitelist'
# Для добавления адресата из режима чтения письма
macro pager \eb '| formail -x From: | listadd >> ~/Mail/blacklist'
macro pager \ew '| formail -x From: | listadd >> ~/Mail/whitelist'
# ---------- Add to black/white list ---------- #
```

Теперь при чтении писем в mutt вам достаточно нажать *Esc–b* для добавления отправителя к «чёрному» списку и *Esc–w* для добавления к «белому». Не добавляйте списки рассылки в «белый» список, т. к. в сообщения из «белого» списка будут складываться в ваш почтовый ящик по умолчанию.

Фильтрация

Теперь поговорим собственно о защите, то есть о фильтрации. Вам придётся дополнить ваш `~/.procmailrc` новыми правилами. Для начала добавьте следующую строку к вашим переменным:

```
SPAM=/home/user_name/Mail/spam          # Файл для писем со спамом
```

Далее добавьте следующие правила перед всеми вашими персональными правилами сортировки.


```
# ---------- Spam Filters ---------- #
# ----------------------
# "Белый"/"Чёрный" список
# ----------------------
# Если отправитель в "белом" списке, то в $DEFAULT.
:0:
* ? formail -x"From" -x"From:" -x"Sender:" \
        -x"Reply-To:" -x"Return-Path:" -x"To:" \
        | egrep -is -f $MAILDIR/whitelist
$DEFAULT

# Если отправитель в "чёрном" списке, то в /dev/null
:0:
* ? formail -x"From" -x"From:" -x"Sender:" \
        -x"Reply-To:" -x"Return-Path:" -x"To:" \
        | egrep -is -f $MAILDIR/blacklist
/dev/null
```

Эффективной практикой борьбы со спамом является фильтрация писем с заголовками вроде «X-Spam-Status: Yes», «X-Spam-Level: 15».

```
# --------------
# Спам-заголовки
# --------------
# Письма с заголовками X-Advertisement
:0:
* ^X-Advertisement:.*
$SPAM

# Письма с установленным заголовком X-Spam-Status
:0:
* ^X-Spam-Status: Yes
$SPAM

# Письма с заголовком X-Spam-Level равным 15 или выше
:0:
* ^X-Spam-Level: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
$SPAM

# Нет получателя
:0:
* To:[ ]*$
$SPAM
```

```
# Пустое поле To:
:0:
* !^To: .*
$SPAM
# ---------- Spam Filters ---------- #
```

Письма, отправитель которых указан в файле `blacklist`, будут утеряны навсегда (отправлены в `/dev/null`). Если вас это не устраивает, то замените `/dev/null` на `$SPAM` или какой-нибудь другой файл.

Надеюсь, вы поняли, что приведённые выше правила не являются каким-либо эталоном. Это лишь то, что помогает мне в борьбе со спамом. Основываясь на этом примере вы сможете сформировать свой личный защитный комплекс, отвечающий вашим запросам.

Не забывайте просматривать ваш файл `$SPAM`, постоянно пополняйте ваши файлы с «белым» и «чёрным» списком — и забудьте о проблеме спама!

Заключение

Мы настроили систему для работы с почтой. Не правда ли, это было совсем не сложно?

Если у вас что-нибудь не получилось, то вы можете задать вопрос в в тематическом списке рассылки или на форуме, посвящённом Linux.

Развитие этого руководства продолжается. Присылайте пожелания по поводу того, каким вы хотите видеть его дальнейшее развитие.

См. также Другие почтовые клиенты для Linux [снаружи, стр. 85]

Linux-класс за час, или X-терминалы, тонкие и ленивые

Георгий Курячий

Терминальный класс? Зачем?

В жизни всегда есть место подвигу — или, по крайней мере, есть место работе, за которую не платят, и которую лучше тебя никто не сделает, а если кто-то сделает её хуже, первым пострадаешь сам.

Пример: развёртывание компьютерного класса из `N` рабочих мест и одного сервера с определённой настройкой того и другого и определённым клиентским и серверным программным наполнением. Как правило, с настройкой сервера справиться нетрудно, особенно подготовив её заранее (взять уже настроенную систему и принести её на съёмном жёстком диске). В конце концов, можно провести пару часов за пультом, устанавливая пакеты и настраивая службы и пользовательское окружение.

Неприятности начинаются, когда встаёшь из-за рабочего стола и понимаешь, что впереди ещё десять-двадцать сеансов установки и настройки системы на рабочие места. Неплохо, если *все* компьютеры-рабочие станции совершенно однотипны, а система имеет режим пакетной установки. Можно просто сделать эталонное рабочее место, а потом растиражировать его. Не дай бог только задумать какое-то изменение на рабочих станциях *после* установки и обнаружить, что пакетного режима для внедрения этих изменений нет.

Собственно, до установки — пакетной или ручной — системы на рабочие станции дело часто и вовсе не доходит, так как ответственный за класс администратор настрого запретил менять что-либо существенное на рабочих местах. Ему тоже пригрезилось двадцать сеансов установки.

В такую переделку попасть легко, стоит только попытаться: проводить спецкурс и лабораторные работы по Linux или по программному продукту для Linux, делать класс доступа к сети Интернет на какой-нибудь конференции, разворачивать площадку для совместной разработки на арендованной технике, использовать широковещательные возможности Linux и т. п.

Между тем, при определённых ограничениях на ресурсы, эта задача решается с помощью X-терминалов[1], называемых также «тонкими клиентами». Смысл решения в том, что на рабочих станциях нужно запустить единственный программный продукт — X-терминал, после чего запускать программы, изменять окружение, работать с файлами и т. д. пользователи будут уже на сервере, а рабочая станция окажется только устройством ввода-вывода («клавиатура–мышь–монитор–сеть»).

Необходимые знания

По сути дела, такая технология не требует *никаких* модификаций программного обеспечения, имеющегося в составе дистрибутива. Удалённый доступ организуется средствами штатной графической подсистемы X11 (в современных дистрибутивах обычно используется реализация X11 по имени `X.Org`), отсюда и название технологии.

Клиент-серверная модель X11

X11 реализует строгое разделение клиентской (прикладной) и серверной (интерфейсной) сторон. **X-сервер** занимается только вводом и выводом: вводом данных с клавиатуры и мыши и выводом графики на видеоадаптер. Запускается X-сервер на рабочей станции — на машине, перед монитором которой сидит пользователь, двигая мышь и нажимая на кнопки. Кроме того, X-сервер выполняет запросы на ввод-вывод, приходящие к нему по сети и другим каналам управления. Стандарт этих запросов называется, естественно, X-протоколом версии 11.

X-клиент — это программа, которая для ввода и вывода использует X-запросы к X-серверу. Никто не мешает X-клиенту, как и любому другому процессу Linux, использовать стандартные потоки ввода-вывода, но X-клиентом он становится, послав запрос серверу. Если клиент и сервер запущены на одной машине, используется специальный файл-дырка (сокет) — обычно он называется `/tmp/.X11-unix/X0`. Если на разных, запросы передаются по сети (порт `6000`)[2].

[1] Обратите внимание, что в этом тексте используется только латинская, а не русская заглавная буква «X».

[2] Последний `0` в имени файла и номере порта означает, что используется сервер номер 0. О нескольких серверах на одной машине см. ниже.

Идентификация

Конечно, не всякий X-клиент может использовать ресурсы X-сервера. Неразумно разрешать кому угодно считывать данные с клавиатуры на рабочей станции: а вдруг в это время вводится пароль? Для доступа к серверу нужен ключ, только зная этот ключ программа (независимо от того, кто и где её запустил) сможет зарегистрироваться на X-сервере и начать обмениваться с ним запросами и ответами. Ключами можно манипулировать с помощью команды `xauth`, которая хранит их в домашнем каталоге пользователя. Таким образом, полученный ключ запоминается с помощью `xauth`, после чего все программы этого пользователя получат право доступа к серверу. В подавляющем большинстве случаев ключ передаётся автоматически: если клиент и сервер запущены одним и тем же пользователем, ключ у него уже есть, а если разными или на разных машинах, ключ переносит программа, организующая сетевое подключение (например, `xdm` или `ssh`).

Обратите внимание на то, что со стороны X-сервера идентификацию можно вообще *отменить* для определённых адресов в сети (например, так: «`xhost + адрес_доверенной_машины`»). Однако такая отмена — большая брешь в безопасности системы (пароли!) и ею можно пользоваться только в отладочных целях. В дистрибутивах ALT Linux сервер вообще не принимает запросов из сети (запускается с параметром «`-nolisten tcp`»); если всё-таки это необходимо, стоит этот параметр в файле `/etc/X11/xinit/xserverrc` закомментировать.

Адресация

«Имеет право доступа» — ещё не значит «подключается и работает». X-клиенту необходимо знать идентификатор X-сервера. Точнее говоря — идентификатор *экрана*, так как X-сервер может предоставлять их несколько (два экрана — типичная ситуация для «двухголовых» видеоадаптеров). Идентификатор экрана в общем случае выглядит так: *адрес*: *номер_сервера.номер_экрана*. *Адрес* — это сетевой адрес компьютера, на котором запущен X-сервер; *номер_сервера* — это номер, под которым сервер зарегистрирован на этом компьютере (как мы уже знаем, по умолчанию это `0`), а *номер_экрана* — номер экрана на этом X-сервере. Чаще всего *адрес* — это либо **доменное имя** компьютера, либо **IP-адрес**, либо ключевое слово «`unix`», означающее, что соединение с сервером можно установить через файл-дырку, не используя

сеть. *Адрес* и *номер_экрана* (вместе с точкой) можно опустить, строка вида «:0» соответствует «unix:0.0».

Большинство программ-клиентов можно запустить с ключом «-display идентификатор_экрана», что позволяет непосредственно направить клиент к определённому X-серверу. Все без исключения X-клиенты используют **переменную окружения** DISPLAY, в которой этот идентификатор можно задать раз и навсегда. Переменную DISPLAY обычно устанавливают системные сценарии и программы, посредством которых пользователь подключается к X-серверу в первый раз, но никто не мешает сделать это из командной строки вручную.

Окновод и рабочий стол

X-сервер сам не делает ничего: он только принимает события от клавиатуры и мыши и передаёт их X-клиенту, которому на данный момент принадлежит **фокус ввода**, а также принимает X-запросы от клиентов и выполняет их (рисует что-нибудь). X-сервер может изменить размер окна X-клиента, переместить его по экрану или над и под окном другого X-клиента, но по своей инициативе этого не делает. Надо, чтобы кто-нибудь (какой-нибудь X-клиент, больше некому) ему это приказал. Для удобства всем этим, а также искусственным переключением фокуса, рисованием красивых рамочек и кнопочек и ещё тысячей разных мелочей занимается специальная программа-окновод — **window manager** (оконный менеджер).

Окноводов для X11 создано несколько десятков. Отличаются они дизайном, поведением, возможностями, размером, способом и глубиной настройки и вообще всем, чем только можно. В дистрибутивах ALT Linux Compact обычно используется всего два — довольно простой и очень небольшой IceWM и kwin, входящий в состав рабочего стола KDE, без которого его запускать особого смысла нет.

Рабочий стол — это уже целая архитектура X-клиентов, общающихся между собой по специальным каналам. Если для IceWM, помимо окновода, запускается ещё пара программ, заставка и системный лоток, то для KDE их необходимо более десятка. Впрочем, и по предоставляемым возможностям они ощутимо различаются. Кстати, никто не мешает, используя IceWM, запускать и программы, являющиеся частями KDE, нужно только помнить, что вместе с такой программой стартует и добрая половина основных служб самого KDE, без которых его приложения не работают.

Диспетчер экранов и сеанс работы

Чтобы не мучить пользователя ручной настройкой DISPLAY и ручным запуском составных частей рабочего стола, запуск всего необходимого заложен в систему сценариев. В частности, запуск всех X-клиентов, которые пользователь хочет видеть после входа в систему, можно оформить в виде командного сценария на sh под именем .xsession или добавлять такие сценарии в подкаталог .xsession.d домашнего каталога пользователя. Запуск сразу всех необходимых для IceWM и KDE X-клиентов уже оформлен в виде сценариев — starticewm и startkde соответственно.

Регистрация в системе непосредственно в графической среде — дело специального класса X-клиентов, именуемых **диспетчерами экранов** (display manager). Таких программ написано несколько, по одной для каждого вида рабочего стола плюс стандартная — xdm. Стандартный xdm довольно беден изобразительно, поэтому обычно используется диспетчер экранов, входящий в состав рабочего стола, например, kdm из KDE. Диспетчер, подобно программе login, спрашивает входное имя и пароль пользователя, после чего запускает выбранный сеанс работы. Kdm, вдобавок, позволяет *выбрать* пользователя из списка и даже использовать для этого собственную фотографию.

Но главное для нас свойство диспетчера экранов — возможность «перекинуть» по сети регистрацию пользователя с одного компьютера на другой. Нетрудно понять, как это делается. Диспетчер, предлагающий регистрацию через сеть, периодически посылает об этом широковещательные сообщения вида «заходите к нам на огонёк». Диспетчер на рабочей станции все такие приглашения регистрирует; полный список можно посмотреть в виде меню, выбрать оттуда нужный удалённый компьютер, кликнуть его («ау! где вы там?») и пожалуйста: диспетчер на рабочей станции сам перезапустит X-сервер с возможностью приёма соединений по сети (параметры «-once -query адрес_компьютера»), а первым X-клиентом у сервера будет диспетчер экранов удалённого компьютера! Относительную безопасность передачи паролей при таком соединении, соблюдение подлинности обеих сторон (а вдруг в процессе перезапуска произошла подмена) и т. п. берёт на себя протокол XDMCP. Для того, чтобы kdm стал рассылать приглашения, необходимо в секции [Xdmcp] настроечного файла /etc/X11/kdm/kdmrc установить настройку Enable в true и перезапустить kdm.

Дистрибутив ALT Linux Compact — возможности Travel CD

Наш пример будет опираться на настройки, имеющие место в дистрибутиве ALT Linux Compact 3.0. Если вы используете какую-нибудь другую версию дистрибутива Linux, задача может решаться и проще, и сложнее, и совсем не решаться. ALT Linux Compact 3.0 вышел в трёх вариациях — дистрибутив для установки на одном CD, так называемый Travel CD и дистрибутив для установки на одном двухслойном DVD, включающий в себя возможности Travel CD и основную часть репозитория стабильных программ ALT Linux Sisyhpus.

Подробнее о Travel CD. Самый известный аналог Compact Travel — дистрибутив Knoppix на базе Debian GNU Linux, авторы которого, а вслед за ними и весь мир, называют подобную технологию Live CD («Живой Диск»). Идея Live CD в том, чтобы работать с Linux даже на тех компьютерах, где Linux нет, а менять что-либо на жёстком диске строго запрещено. Из положения можно выйти, установив все нужные программы на CD заранее (такое устройство будет, конечно, доступно только на чтение), а для записи файлов завести небольшую виртуальную файловую систему в памяти. Останется только в конце работы сложить нужные файлы на дискету или flash-диск, либо скопировать их куда-нибудь по сети. К жёсткому диску при этом система вообще не обращается, хотя при необходимости его можно использовать.

Две главные трудности Live CD — разнообразие аппаратуры и нехватка ресурсов. Для того чтобы одна и та же операционная система с одного и того же диска могла работать практически на любом компьютере, необходимо помнить множество различных настроек различных устройств и уметь их автоматически подбирать. С другой стороны, как бы это ни было сложно, именно этим занимается программа-установщик системы, и его части по-умному внедрены в Live CD. Впрочем, всегда остаётся слой компьютеров (как правило — самых старых и самых новых), на котором Live CD уже или ещё не запускается.

Касаемо нехватки ресурсов, именно — оперативной памяти. Нехватка возникает от того, что нельзя ничего записывать на жёсткий диск. Обычная Linux-система создаёт на жёстком диске специальный раздел (swap), на который выгружаются временно не используемые страницы оперативной памяти. Пока запускаемые пользователем программы вместе помещаются в физически имеющуюся оперативную память. Хорошо, если нет — та часть памяти, что дольше всего не использовалась, выгружается на диск, освобождая дополнительное

место. Когда эти страницы памяти на самом деле понадобятся программе, выгрузится что-нибудь ещё. Только для Live CD такой раздел завести негде. В Knoppix внедрены некоторые дополнительные средства, позволяющие смонтировать файловую систему на диске и разместить область подгрузки в файле, но это — уже прямое нарушение заповеди «ничего не трогай». В ALT Linux Compact Travel автоматических средств для этого нет, хотя вручную, конечно, можно сделать что угодно.

Если вам вздумается без конца запускать разнообразные программы, пожирающие память, то через какое-то время вы заметите, что каждый следующий запуск идёт намного медленнее предыдущего. Это она — загрузка и выгрузка страниц. Не то для Travel CD: выгружать некуда, и с окончанием свободного места в памяти программы попросту перестанут запускаться.

Есть и ещё один недостаток всех Live CD: скорость чтения данных с лазерного диска на порядки ниже скорости чтения данных с винчестера (жёсткого диска). Особенно это касается работы не с потоками данных (для которых лазерный диск был разработан), а с настоящей файловой системой, когда считывающей головке лазерного привода приходится метаться от начала к концу диска и обратно. Этому горю помочь нельзя — если, конечно, вы не собираетесь использовать терминальный класс, в котором с CD загружается только часть системы и X-сервер, а вся работа ведётся на «нормальной» удалённой Linux-машине.

Варианты терминального класса

Нулевой вариант

Для начала стоит решить для себя: а нужна ли вообще вся эта возня с X-терминалами? В самом деле, если вся задача — запустить сетевой навигатор или почтовую программу, то проще всего наплодить Travel CD и работать именно с ними. Медленную скорость загрузки `Firefox`, `Thunderbird` и даже `OpenOffice.org` можно стерпеть — тем более, что, однажды загрузившись, программы начинают работать довольно быстро.

Однако в случае, когда работа предстоит разнообразная, требующая дополнительной настройки персональных рабочих мест, запуска различных программ и т. п., недостатки технологии Live дают о себе

знать, и терминальный класс представляется логичной и несложной заменой классу с Travel CD.

Тренировочный вариант: сервер приложений

Вооружившись полученными знаниями, построим терминальный класс... вообще не устанавливая Linux, из двух компьютеров с двумя Travel CD. Одна машина будет сервером приложений, другая — X-терминалом. На самом деле это потребует даже *больше* действий по сравнению со стандартной ситуацией: Linux-сервер с *установленной* системой и произвольный клиент, загружаемый с Travel CD.

Сначала изготовим сервер приложений. Загрузим первую попавшуюся машину с Travel CD. Вместо диспетчера экранов нам предлагается т. н. autologin — автоматический вход в систему под именем `altlinux` и без пароля. Автоматический вход надо отменить, для чего следует перейти в системную консоль (`Ctrl-Alt-F1`[1]), зарегистрироваться как суперпользователь `root` (у него тоже нет пароля! а зачем пароль, когда ничего испортить нельзя?) и удалить файл `/etc/sysconfig/autologin`:

```
[root@Compact root]# rm -f /etc/sysconfig/autologin
[root@Compact root]# killall X
```

Команда `killall X` посылает сигнал экстренной остановки X-серверу (который так и называется — `X`). Система настроена так, что X-сервер запустится заново, но теперь вместо автоматической регистрации будет использована обычная — `kdm`. Другие Live-дистрибутивы могут применять какой-нибудь иной механизм, например, непосредственный запуск X-сервера от имени пользователя или авторегистрацию средствами диспетчера экранов (соответствующие настройки `kdm` так и называются — `AutoLoginEnable` и `AutoLoginUser`); здесь мы ограничимся вариантом ALT Linux Compact 3.0.

Последовательность клавиш для возврата в графический режим зависит от того, какую системную консоль захватывает графическая оболочка. Для дистрибутивов ALt Linux эта консоль — седьмая, и переход в графический режим, соответственно, `Ctrl-Alt-F7`.

Вторая задача — запустить `kdm` в широковещательном режиме, чтобы пользователи могли регистрироваться на этой машине дистанционно. Как сказано выше, для этого нужно отредактировать файл

[1]Если после `Ctrl-Alt-F1` в Compact 3.0 Travel вместо системной консоли вы видите индикатор загрузки системы — зелёную картинку с пингвинами — просто нажмите `Esc`.

`/etc/X11/kdm/kdmrc`, заменив в секции `[Xdmcp]` строку `Enable=false` на `Enable=true`. Делать это надо, разумеется, с правами суперпользователя, например, из той же системной консоли. Если вы *не* умеете обращаться с редактором `vim`, воспользуйтесь `mcedit -a /etc/X11/kdm/kdmrc`[1]. В нашем примере мы воспользуемся тем, что строка «`Enable=false`» в этом файле одна, так что её можно легко изменить пакетным текстовым редактором `sed` прямо из командной строки:

```
[root@Compact root]# sed -i~ -e 's/^Enable=false/Enable=true/'
/etc/X11/kdm/kdmrc
[root@Compact root]# diff /etc/X11/kdm/kdmrc /etc/X11/kdm/kdmrc~
 112c112
 < Enable=true
 ---
 > Enable=false
[root@Compact root]# service dm restart
 Stopping display manager service:                          [ DONE ]
 Starting display manager service:                          [ DONE ]
```

После изменения настроечного файла `kdm` следует перезапустить — либо уже нам известным «грубым» путём, экстренно остановив текущий X-сервер, либо так, как показано на примере: штатной командой перезапуска службы «**d**isplay **m**anager» `service dm restart`. В примере также показано, как с помощью команды `diff` убедиться, что замена произошла.

Сервер приложений готов. Заметим, что на нём можно было вообще не запускать X-сервера (графической среды), один только широковещательный диспетчер экранов, но это потребовало бы бóльших трудов и ручной настройки.

Тренировочный вариант: рабочая станция

Всё, что необходимо сделать на рабочей станции, это загрузить её с Travel CD, отменить авторегистрацию (уже известным способом) и выбрать в меню пункт «Удалённый вход» (рис. 9.1).

Если `kdm` на сервере приложений уже широковещает, и оба компьютера подключены к одной локальной сети, то список удалённых подключений будет непуст (рис. 9.2).

Осталось только выбрать нужный компьютер из списка принимающих удалённое подключение и... увидеть стандартное приглашение

[1]В системной консоли Compact 3.0 файловый менеджер `mc` и его редактор надо запускать с ключом `-a`, чтобы он не пытался рисовать псевдографику.

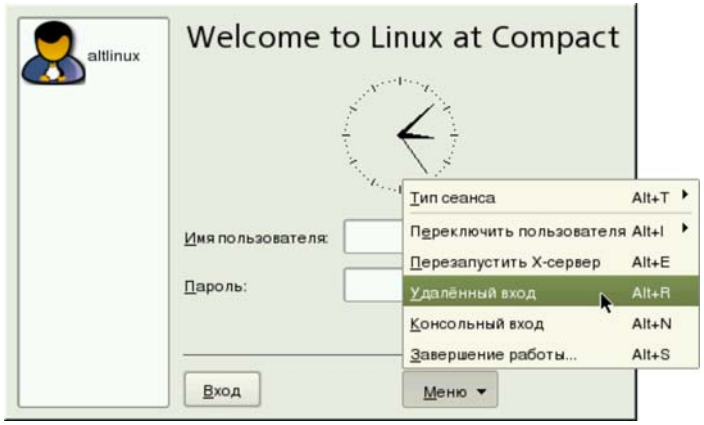

Рис. 9.1. Подключение к серверу приложений с рабочей станции (Travel CD)

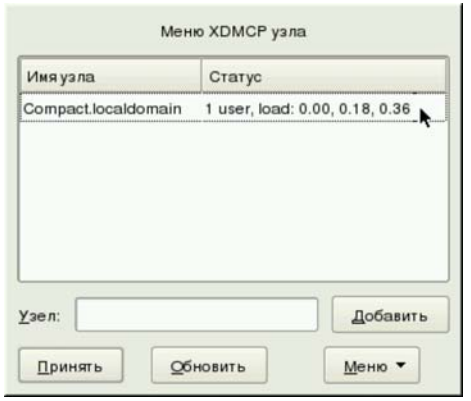

Рис. 9.2. Список удалённых серверов приложений, принимающих подключения

`kdm`! Пусть вас не обманет сходство: это приглашение, как и весь последующий сеанс работы, запущено с удалённой машины. Проверить это легче лёгкого: зарегистрироваться с рабочей станции (у пользователя `altlinux` нет пароля) и запустить команду `who` в консоли на удалённой машине:


```
[root@Compact root]# who
 root     tty1         Feb 12 14:07 (localhost)
 altlinux 169.254.112.94:0 Feb 12 14:22 (169.254.112.94)
```

Это был «тренировочный» способ организации терминального класса «из ничего». Применений на практике ему не так уж много — разве что в классе, где все машины, кроме одной, совсем плохонькие, зато эта одна — мощнее некуда. Программы с сервера приложений, изготовленного из Travel CD, запускаются точно так же медленно, а ресурсы кончаются намного быстрее — сообразно количеству работающих пользователей.

Ленивый вариант

В действительности нужен один компьютер с *установленной* операционной системой — например, с тем же ALT Linux Compact 3.0. Желательно на этом компьютере иметь побольше оперативной памяти и завести swap-область приличного размера. Как это сделать — написано в инструкции по установке, здесь повторяться не стоит. Для работы пяти-семи человек достаточно стандартной установки, а оперативной памяти должно быть от 64 до 128 мегабайтов на пользователя. Установка Compact 3.0 по умолчанию *не* включает автоматическую регистрацию, поэтому всё, что нужно — это включить широковещательный режим `kdm`, как показано выше. «Ленивый сервер» готов.

А вот «ленивый клиент» готов заранее: в меню загрузки ALT Linux Compact 3.0 Travel уже есть пункт «X Terminal»! При выборе этого пункта запускается только графическая подсистема и заранее перенастроенный `kdm` в режиме удалённого входа, так что ничего делать не надо.

Можно работать.

Что хотелось бы иметь на сервере, или Разумный вариант

Ленивый вариант терминального класса отличается от тренировочного тем, что администратор облегчает себе жизнь, пользуясь подготовленными настройками системы. В то же время оба варианта похожи: в них не предусмотрено облегчить жизнь *пользователя*: бери, что дают. А то, что дают в *офисном* дистрибутиве, выглядит недостаточным для полноценной работы сервера. В первую очередь дополнительной настройки требуют подсистемы учётных записей и автоматической настройки рабочих станций.

Разумный вариант: пользователи класса

Если сервер — результат установки Compact 3.0 по умолчанию, на нём будет заведена только одна пользовательская учётная запись. Можно оставить и так, предложив пользователям X-терминалов регистрироваться одинаково, однако это неудобно: начнутся споры на тему «чей файл», «кто перенастроил мою почту» и т. п. Намного правильней завести каждому пользователю свою учётную запись с собственным паролем, окружением и т. п., благо Linux это с лёгкостью позволяет. В ALT Linux можно воспользоваться программой «ALT Linux control center» (команда `acc`), либо завести их вручную с помощью, например, такого сценария:

```
[root@Compact root]# for N in alex george nik; do useradd $N; done
[root@Compact root]# passwd george
 passwd: updating all authentication tokens for user george.

 You can now choose the new password or passphrase.
  . . .
 Enter new password:
 Re-type new password:
 passwd: all authentication tokens updated successfully.
```

В примере создаётся три учётные записи с входными именами `alex`, `george` и `nik`. Затем каждого пользователя надо попросить ввести пароль с помощью команды `passwd входное_имя`. Если этого не сделать, зарегистрироваться с такой учётной записью будет невозможно.

Теперь диспетчер экранов `kdm` покажет список заведённых пользователей, а в пунктах меню — возможные варианты рабочих столов. Для экономии памяти на сервере приложений (всё-таки *все* программы запускаются там), можно порекомендовать пользователям выбирать пункт «`IceWM`» (рис. 9.3).

Системный администратор может установить и другие оконные менеджеры, например, WindowMaker или FVWM2, и настроить их.

Разумный вариант: автоматическая сетевая настройка и доменные имена

При старте Compact Travel опрашивает имеющиеся сетевые устройства и присваивает им адреса из внутреннего (intranet) диапазона (`169.254.0.0` — `169.254.255.254`), предварительно проверяя, не занят

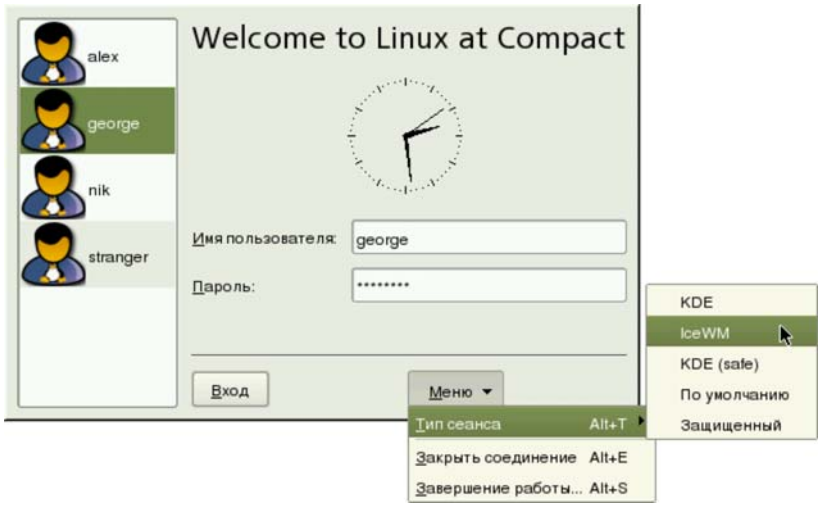

Рис. 9.3. Выбор оконного менеджера IceWM

ли соответствующий адрес. Предполагается, что среди более, чем шестидесяти тысяч адресов найдётся свободный. Однако, если в локальной сети имеется служба автоматической настройки абонентов DHCP (**d**ynamic **h**ost **c**onfiguring **p**rotocol), система воспользуется ею: в этом случае заработает и доступ в Интернет, и преобразование доменных имён (DNS), что сделает работу намного удобнее.

Если служб DHCP и DNS в локальной сети нет, их несложно организовать. Для этого необходим «большой» дистрибутив ALT Linux — например ALT Linux Compact 3.0 DVD (серверные программы не включены в CD-версию дистрибутива). Для DHCP рекомендуется воспользоваться пакетом `dhcp-server`, а для DNS — `pdnsd`.

Пакет `dhcp-server` содержит демон `dhcpd` и практически не имеет альтернативных реализаций. Настраивать его надо редактированием файла `/etc/dhcp/dhcpd.conf` и запуском службы `dhcpd`. Пакет `pdnsd` — небольшая и очень простая замена более мощному и сложному `bind`. Сам `pdnsd` применяется, в основном, для кеширования DNS-запросов при работе на очень тонких линиях связи (например, по модему), но его легко превратить в локальный DNS-сервер, отредактировав файл `/etc/hosts` и указав этот файл в качестве источника в `/etc/pdnsd.conf`. Для более подробного знакомства с настройкой `dhcpd` и `pdnsd`

обращайтесь к руководствам по ним, которые поставляются в ALT Linux в составе пакетов, и не забывайте, что в дистрибутивах ALT Linux файл `/etc/hosts` необходимо распространять в изолированные окружения командой `update_chrooted conf`.

Недостатки терминального класса и борьба с ними

Поход за дискетой

Главный недостаток терминального класса на основе «тонких» клиентов (с запуском всех приложений на сервере) проявится сразу после того, как кто-либо из пользователей захочет переписать данные на дискету или flash-диск. Несмотря на то, что окружение у каждого пользователя X-терминала своё, внешние устройства всё равно общие, и устройства эти подключены к *серверу*, а не к рабочей станции — ведь все программы запускаются на сервере! Чтобы иметь доступ ко внешним устройствам на локальном компьютере, надо и регистрироваться локально, а не по сети... с чего начали, тем и закончили?

Есть несколько способов обойти этот недостаток, не исправляя. Во-первых, можно возложить на *администратора* работу по копированию пользовательских файлов на пользовательские дискеты: файлы-то все на одной машине, в домашних каталогах пользователей (`/home/входное_имя/`). Перед началом и после окончания работы к администратору выстраивается небольшая очередь с дискетами и flash-дисками — и это разумное решение, например, для учебного класса, где копирование файлов с системы на систему — ситуация нештатная и требующая контроля. Ведь за какой X-терминал ученик ни сядет, доступ к своему домашнему каталогу ему гарантирован, зачем тогда дискета?

Во-вторых, никто не запрещает регистрироваться локально на X-терминалах; это можно сделать, например, с системной консоли (`Ctrl-Alt-F1`). После чего уже сравнительно просто передать файл с удалённой машины на внешнее устройство рабочей станции. Для этого на удалённой машине должен быть запущен сервис Secure Shell (`sshd`), а на рабочей станции — установлен клиент `ssh` (точнее, `scp`, Secure Copy):

```
[root@Compact root]# scp george@169.254.56.249:file /media/floppy/

 The authenticity of host '169.254.56.249' can't be established.
 RSA key fingerprint is
9d:17:85:40:4c:cf:bb:e3:57:ec:86:35:69:bf:33:7a.
 Are you sure you want to continue connecting (yes/no)? yes
 Warning: Permanently added '169.254.56.249' (RSA) to the list of
known hosts.
 george@169.254.56.249's password:
 file                                  100%   29     0.3KB/s   00:00
```


Конечно, только что народившаяся рабочая станция не может знать ssh-ключ удалённой машины, поэтому `scp` предлагает его проверить («Are you sure you want to continue connecting...»). После того, как пользователь один раз ответит «yes», ключ запоминается, и передача файлов требует лишь пароля.

В-третьих, можно применить другую технологию совместной работы, отличную от описанных выше «ленивых и тонких» X-терминалов. Создание таких терминальных классов, в зависимости от типа дистрибутива, может происходить и полуавтоматически, и совсем вручную. Дистрибутивы ALT Linux Compact, строго говоря, не предназначены для серверных разработок, хотя версия 3.0 DVD включает все необходимые составные части.

Двадцать лазерных дисков

Ещё одно неудобство терминального класса на базе любого Live CD: Live CD. Вернее, их количество и необходимость с них загружаться. Довольно быстро возникает идея: большинство компьютеров умеют загружаться по сети. Ну так пусть по сети на них загружается *то же самое*, что и на Live CD — скорость работы повысится, двадцать Live CD отправятся в сейф.

К сожалению, приходится признать, что сформулировать задачу намного проще, чем решить. Для организации сетевой загрузки необходимо:

- настроить службы DHCP и TFTP;
- подобрать конфигурацию загрузчика PXE (например, `pxelinux` из пакета `syslinux`);
- решить, куда размещать «то же самое» — содержимое Live CD (раньше оно было на CD, а теперь? в памяти целиком? в сети?);

- самостоятельно сформировать стартовую операционную систему, которая, собственно, загружается по сети, добывает содержимое Live CD и начинает работать, как ни в чём не бывало.

Последний пункт — самый сложный, требует дополнительного знания сразу нескольких областей (сетевое взаимодействие, архитектура Linux, особенности стартовой системы конкретного дистрибутива и т. п.). Перед Compact 3.0 эта задача даже и не ставилась. Видимо, сетевая загрузка «Live» — дело будущего.

Другие подходы

«Толстые клиенты»

Кроме носителей данных, бывают иные внешние устройства, в первую очередь — мультимедийные, пользоваться которыми можно только напрямую. Просмотр фильма с удалённого компьютера мало того, что загружает сеть распакованным видеопотоком — по сути, последовательностью кадров — он не использует аппаратные возможности видеоадаптера, связанные с обработкой *исходного* формата (например, аппаратное декодирование MPEG). А если запустить на одной машине сразу десять видеопроигрывателей, совсем не очевидно, что и она, и локальная сеть хорошо справятся с такой нагрузкой. Другое дело, если проигрыватель запущен на рабочей станции, а файл с фильмом (который в десятки раз меньше потока кадров) добывается с удалённого компьютера.

Ещё курьёзнее ситуация со звуком. Если запустить аудиопроигрыватель на удалённой машине, что запищит? Скорее всего, аудиоколонки на удалённой машине (если они там есть). Самый простой выход: аудиопроигрыватель надо запускать прямо на рабочей станции. И видеопроигрыватель тоже: кому нужны фильмы без звука?

Можно решить эту задачу и так: запустить на рабочей станции аудио-сервер (аналог X-сервера для звуковых данных, например, `arts` или `esd`), а на удалённой машине запускать только те звуковые программы, что умеют с этим сервером работать. Останется только стандартным для данного звукового сервера способом сообщить этим программам, где он находится.

Так или иначе, представление о рабочем месте, как о клавиатуре, мыши и графическом дисплее, стоящем на тонкой и плоской коробочке, в которой работает только X-сервер, меняется. Теперь эта

коробочка (она называется «компьютер») заметно толстеет: на ней запускаются некоторые X-клиенты — а стало быть, на ней должно, хотя бы отчасти, воспроизводиться пользовательское окружение, быть доступен домашний каталог и т. п.

«Толстый клиент» гораздо требовательнее к оперативной памяти рабочей станции. Проще всего сделать т. н. «бездисковый клиент» — рабочую станцию, поведение которой не отличается от обычной Linux-станции, за исключением того, что загрузка её происходит по сети, и все файловые системы монтируются тоже по сети, а жёсткий диск не используется. В этом случае все X-клиенты (в том числе и весь рабочий стол) будут запускаться с рабочей станции. Swap-область такой машине просто необходима, её принято перенаправлять в файл, располагающийся на сервере, то есть опять-таки через сеть (swap через сеть работает очень, очень медленно и может съесть всю её пропускную способность).

По всей вероятности, можно организовать класс бездисковых клиентов с помощью *модифицированного* Compact 3.0 Travel: CD-часть будет определять и настраивать сеть, монтировать сетевые диски, после чего система будет вести себя так, как это ей предписывается заранее разработанными сценариями на сервере. По сути дела, на сервере нужна целая схема поддержки бездисковых клиентов. ALT Linux Compact — *пользовательский* дистрибутив, этой схемы не имеющий, так что и режима «толстого клиента» в Travel CD по умолчанию нет.

Можно попытаться организовать «толсто-тонкий» клиент: все приложения, кроме некоторых, запускать с сервера, а этим некоторым организовать *доступ* к данным на сервере. Или «тонко-толстый»: оставить на рабочей станции только X-сервер да проигрыватели, так что пользователь, волей-неволей, все остальные программы будет запускать всё-таки с сервера. Этому может помочь вот какое соображение: программа терминального доступа Secure Shell (`ssh`), с помощью которой пользователь будет заходить на сервер, обладает свойством перенаправления некоторых сетевых соединений, в частности — X-запросов. В результате удалённо запущенные X-клиенты без дополнительных настроек передают X-запросы локальному X-серверу, минуя процедуру идентификации или небезопасного применения `xhost`.

Перенаправление X-запросов происходит так. Команду `ssh` надо выполнить с ключом `-X`; тогда сервер Secure Shell (`sshd`) на удалённой машине зарегистрирует там *виртуальный* X-сервер (обычно под номером `10`, а если такой уже есть — под следующим незанятым) и устанавливает переменную окружения `DISPLAY`. X-клиент спокойно

обращается к этому новому X-серверу, не подозревая о том, что все X-запросы **sshd** передаёт по установленному соединению, а **ssh**, сам будучи X-клиентом, ретранслирует их от своего имени уже на машине пользователя:

```
anyuser@workstation:~> echo $DISPLAY
 :0.0
anyuser@workstation:~> glxinfo
 name of display: :0.0
 display: :0  screen: 0
 direct rendering: Yes
 server glx vendor string: SGI
  . . .
anyuser@workstation:~> ssh -X george@linuxserver.some.domain
 george@linuxserver.some.domain's password:
 Last login: Mon Feb 13 13:59:22 2006 from localhost.localdomain
george@linuxserver:~> echo $DISPLAY
 localhost:10.0
george@linuxserver:~> glxinfo
 name of display: localhost:10.0
 display: localhost:10  screen: 0
 direct rendering: Yes
 server glx vendor string: SGI
  . . .
```

Всё бы ничего, только обычному пользователю управляться со смешанным клиентом будет непросто: надо всё время держать в уме, что некоторые программы запускаются «там», а некоторые — «тут», что файлы, доступные «там» не всегда видны «тут» и наоборот, что программа, запущенная «тут» потребляет системные ресурсы, а «там» — сетевые... Ситуацию работы на *двух* машинах одновременно упростить, увы, нельзя.

Linux Terminal Server Project

Выше речь была, в основном, о возможностях организации терминального класса на базе ALT Linux Compact 3.0, о том, как решать такую задачу *дистрибутивно* (без таких экзотических упражнений, как пересборка ядра и доработка программного обеспечения).

Но есть и другой подход. Целая команда разработчиков всего мира поддерживает проект Linux Terminal Server[1], в котором они как раз

[1] http://www.ltsp.org

и занимаются доработкой и пересборкой базовых компонентов Linux для того, чтобы решить основную задачу: организацию удобного в эксплуатации терминального класса на любом оборудовании.

Создатели LTSP устраняют возникающие затруднения естественным для Linux путём: дописывают отсутствующие в программном обеспечении возможности и создают инфраструктуру для их применения. LTSP — «дистрибутив в дистрибутиве». Он требует некоторых усилий по интеграции в базовую систему и содержит всё необходимое для запуска на рабочей станции. В LTSP решены задачи и доступа к локальным внешним устройствам, и разделения приложений на удалённые и локальные, и воспроизведения звука, и многие другие. Исследовать вопрос поглубже можно на русскоязычном сайте[1] проекта, стоит только отметить, что, несмотря на хорошую документацию и массу успешных установок, развёртывание LTSP-класса продолжает оставаться делом квалифицированного системного администратора.

См. также Графическая подсистема X11 . . . . . [стр. 126]
Дистрибутив ALT Linux 3.0 Compact (и его версия TravelCD) . . . . . [снаружи, стр. 12]

[1] http://www.ltsp.ru

Книги издательского дома «ДМК-пресс» можно заказать в торгово-издательском холдинге «Альянс-книга» наложенным платежом, выслав открытку или письмо по почтовому адресу: 123242, Москва, а/я 20 или по электронному адресу **post@abook.ru**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и адрес получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в Internet-магазине: **www.abook.ru**.

Оптовые покупки: тел. (495) 258-91-94, 258-91-95;
электронный адрес **abook@abook.ru**.

Подписано в печать 30.06.06. Формат $60 \times 90\ ^1/_{16}$
Гарнитура «Квант Антиква». Печать офсетная
Усл. печ. л. 26. Тираж 3000 экз. Заказ №

Издательство «ДМК-пресс», 123007, Москва, 1-й Силикатный пр-д, 14
Электронный адрес: **www.dmk-press.ru**
Электронная почта: **books@dmk-press.ru**